MPASM™ Assembler,
MPLINK™ Object Linker
MPLIB™ Object Librarian
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in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications.

Microchip makes no representations or warranties of any kind, whether express or implied, written or oral, statutory or otherwise, related to the information, including but not limited to its condition, quality, performance, merchantability or fitness for purpose. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KeelLog, KeelLog logo, MPLAB, PIC, PICmicro, PICSTART, rFIC, SmartShunt and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, nanoWatt XLP, PICKit, PICDEM, PICDEM.net, PICtail, PIC[32] logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rFLAB, Select Mode, Total Endurance, TSHARC, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

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

© 2009, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company’s quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KeelLog® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001:2000 certified.
# Table of Contents

Preface ....................................................................................................................... 1  
PIC1X MCU Language Tools and MPLAB IDE ............................................................ 9  
Part 1 – MPASM Assembler  

Chapter 1. MPASM Assembler Overview  
1.1 Introduction .................................................................................................... 23  
1.2 MPASM Assembler Defined .......................................................................... 23  
1.3 How MPASM Assembler Helps You ............................................................ 23  
1.4 Assembler Migration Path ............................................................................ 23  
1.5 Assembler Compatibility Issues ................................................................... 24  
1.6 Assembler Operation .................................................................................... 24  
1.7 Assembler Input/Output Files ....................................................................... 26  

Chapter 2. Assembler Interfaces  
2.1 Introduction ................................................................................................... 33  
2.2 MPLAB IDE Interface ................................................................................... 33  
2.3 Windows Interface ........................................................................................ 34  
2.4 Command Line Interface .............................................................................. 35  

Chapter 3. Expression Syntax and Operation  
3.1 Introduction ................................................................................................... 37  
3.2 Text Strings .................................................................................................. 37  
3.3 Reserved Words and Section Names .......................................................... 39  
3.4 Numeric Constants and Radix ...................................................................... 39  
3.5 Arithmetic Operators and Precedence ......................................................... 40  

Chapter 4. Directives  
4.1 Introduction .................................................................................................... 43  
4.2 Directives by Type ........................................................................................ 43  
4.3 access_ovr - Begin an Object File Overlay Section in Access  
RAM (PIC18 MCUs) ......................................................................................... 46  
4.4 __badram - Identify Unimplemented RAM ................................................. 46  
4.5 __badrom - Identify Unimplemented ROM ............................................... 47  
4.6 bankisel - Generate Indirect Bank Selecting Code  
(PIC12/16 MCUs) ............................................................................................... 48  
4.7 banksel - Generate Bank Selecting Code .................................................. 50  
4.8 cblock - Define a Block of Constants ......................................................... 52  
4.9 code - Begin an Object File Code Section .................................................. 54  
4.10 code_pack - Begin an Object File Packed Code Section  
(PIC18 MCUs) ..................................................................................................... 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.11</td>
<td>__config - Set Processor Configuration Bits</td>
<td>55</td>
</tr>
<tr>
<td>4.12</td>
<td>config - Set Processor Configuration Bits (PIC18 MCUs)</td>
<td>57</td>
</tr>
<tr>
<td>4.13</td>
<td>constant - Declare Symbol Constant</td>
<td>58</td>
</tr>
<tr>
<td>4.14</td>
<td>da - Store Strings in Program Memory (PIC12/16 MCUs)</td>
<td>59</td>
</tr>
<tr>
<td>4.15</td>
<td>data - Create Numeric and Text Data</td>
<td>60</td>
</tr>
<tr>
<td>4.16</td>
<td>db - Declare Data of One Byte</td>
<td>62</td>
</tr>
<tr>
<td>4.17</td>
<td>de - Declare EEPROM Data Byte</td>
<td>64</td>
</tr>
<tr>
<td>4.18</td>
<td>#define - Define a Text Substitution Label</td>
<td>65</td>
</tr>
<tr>
<td>4.19</td>
<td>dt - Define Table (PIC12/16 MCUs)</td>
<td>67</td>
</tr>
<tr>
<td>4.20</td>
<td>dtm - Define Table (Extended PIC16 MCUs Only)</td>
<td>67</td>
</tr>
<tr>
<td>4.21</td>
<td>dw - Declare Data of One Word</td>
<td>68</td>
</tr>
<tr>
<td>4.22</td>
<td>else - Begin Alternative Assembly Block to if Conditional</td>
<td>68</td>
</tr>
<tr>
<td>4.23</td>
<td>end - End Program Block</td>
<td>69</td>
</tr>
<tr>
<td>4.24</td>
<td>endc - End an Automatic Constant Block</td>
<td>69</td>
</tr>
<tr>
<td>4.25</td>
<td>endif - End Conditional Assembly Block</td>
<td>70</td>
</tr>
<tr>
<td>4.26</td>
<td>endm - End a Macro Definition</td>
<td>70</td>
</tr>
<tr>
<td>4.27</td>
<td>endw - End a while Loop</td>
<td>71</td>
</tr>
<tr>
<td>4.28</td>
<td>equ - Define an Assembler Constant</td>
<td>71</td>
</tr>
<tr>
<td>4.29</td>
<td>error - Issue an Error Message</td>
<td>72</td>
</tr>
<tr>
<td>4.30</td>
<td>errorlevel - Set Message Level</td>
<td>73</td>
</tr>
<tr>
<td>4.31</td>
<td>exitm - Exit from a Macro</td>
<td>75</td>
</tr>
<tr>
<td>4.32</td>
<td>expand - Expand Macro Listing</td>
<td>77</td>
</tr>
<tr>
<td>4.33</td>
<td>extern - Declare an Externally Defined Label</td>
<td>78</td>
</tr>
<tr>
<td>4.34</td>
<td>fill - Specify Program Memory Fill Value</td>
<td>80</td>
</tr>
<tr>
<td>4.35</td>
<td>global - Export a Label</td>
<td>82</td>
</tr>
<tr>
<td>4.36</td>
<td>idata - Begin an Object File Initialized Data Section</td>
<td>82</td>
</tr>
<tr>
<td>4.37</td>
<td>idata_acs - Begin an Object File Initialized Data Section in Access RAM</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>(PIC18 MCUs)</td>
<td></td>
</tr>
<tr>
<td>4.38</td>
<td>__idlocs - Set Processor ID Locations</td>
<td>85</td>
</tr>
<tr>
<td>4.39</td>
<td>if - Begin Conditionally Assembled Code Block</td>
<td>86</td>
</tr>
<tr>
<td>4.40</td>
<td>ifdef - Execute If Symbol has Been Defined</td>
<td>88</td>
</tr>
<tr>
<td>4.41</td>
<td>ifndef - Execute If Symbol has not Been Defined</td>
<td>89</td>
</tr>
<tr>
<td>4.42</td>
<td>#include - Include Additional Source File</td>
<td>90</td>
</tr>
<tr>
<td>4.43</td>
<td>list - Listing Options</td>
<td>91</td>
</tr>
<tr>
<td>4.44</td>
<td>local - Declare Local Macro Variable</td>
<td>92</td>
</tr>
<tr>
<td>4.45</td>
<td>macro - Declare Macro Definition</td>
<td>94</td>
</tr>
<tr>
<td>4.46</td>
<td>__maxram - Define Maximum RAM Location</td>
<td>95</td>
</tr>
<tr>
<td>4.47</td>
<td>__maxrom - Define Maximum ROM Location</td>
<td>96</td>
</tr>
<tr>
<td>4.48</td>
<td>messg - Create User Defined Message</td>
<td>96</td>
</tr>
<tr>
<td>4.49</td>
<td>noexpand - Turn off Macro Expansion</td>
<td>98</td>
</tr>
<tr>
<td>4.50</td>
<td>nolist - Turn off Listing Output</td>
<td>98</td>
</tr>
<tr>
<td>4.51</td>
<td>org - Set Program Origin</td>
<td>98</td>
</tr>
<tr>
<td>4.52</td>
<td>page - Insert Listing Page Eject</td>
<td>101</td>
</tr>
<tr>
<td>4.53</td>
<td>pagesel - Generate Page Selecting Code (PIC10/12/16 MCUs)</td>
<td>102</td>
</tr>
</tbody>
</table>
4.54 `pageselw` - Generate Page Selecting Code Using WREG Commands (PIC10/12/16 MCUs) ......................................................... 103
4.55 `processor` - Set Processor Type ......................................................... 104
4.56 `radix` - Specify Default Radix ............................................................ 105
4.57 `res` - Reserve Memory ....................................................................... 106
4.58 `set` - Define an Assembler Variable .................................................. 108
4.59 `space` - Insert Blank Listing Lines .................................................... 109
4.60 `subtitle` - Specify Program Subtitle ................................................... 109
4.61 `title` - Specify Program Title .............................................................. 110
4.62 `udata` - Begin an Object File Uninitialized Data Section ..................... 110
4.63 `udata_acs` - Begin an Object File Access Uninitialized Data Section (PIC18 MCUs) .............................................................. 111
4.64 `udata_ovr` - Begin an Object File Overlaid Uninitialized Data Section ......................................................................................... 113
4.65 `udata_shr` - Begin an Object File Shared Uninitialized Data Section (PIC12/16 MCUs) .............................................................. 115
4.66 `#undefine` - Delete a Substitution Label ............................................ 116
4.67 `variable` - Declare Symbol Variable .................................................. 117
4.68 `while` - Perform Loop While Condition is True .................................. 118

**Chapter 5. Assembler Examples, Tips and Tricks**

5.1 Introduction ............................................................................................ 121
5.2 Example of Displaying Count on Ports .................................................. 122
5.3 Example of Port B Toggle and Delay Routines ....................................... 123
5.4 Example of Calculations with Variables and Constants ......................... 130
5.5 Example of a 32-Bit Delay Routine ....................................................... 132
5.6 Example of SPI Emulated in Firmware .................................................. 134
5.7 Example of Hexadecimal to ASCII Conversion ..................................... 136
5.8 Other Sources of Examples .................................................................... 137
5.9 Tips and Tricks ....................................................................................... 137

**Chapter 6. Relocatable Objects**

6.1 Introduction ............................................................................................ 141
6.2 Header Files ........................................................................................... 141
6.3 Program Memory .................................................................................... 142
6.4 Low, High and Upper Operators ............................................................ 142
6.5 RAM Allocation ..................................................................................... 145
6.6 Configuration Bits and ID Locations ...................................................... 146
6.7 Accessing Labels From Other Modules ............................................... 146
6.8 Paging and Banking Issues ..................................................................... 147
6.9 Generating the Object Module ............................................................... 148
6.10 Code Example ....................................................................................... 148
Chapter 7. Macro Language

7.1 Introduction ................................................................................................. 151
7.2 Macro Syntax ............................................................................................. 151
7.3 Macro Directives Defined ........................................................................... 152
7.4 Macro Definition ......................................................................................... 152
7.5 Macro Invocation ........................................................................................ 152
7.6 Macro Code Examples ............................................................................... 153

Chapter 8. Errors, Warnings, Messages, and Limitations

8.1 Introduction ................................................................................................. 155
8.2 Assembler Errors ........................................................................................ 155
8.3 Assembler Warnings .................................................................................. 162
8.4 Assembler Messages ................................................................................. 165
8.5 Assembler Limitations ................................................................................ 167

Part 2 – MPLINK Object Linker

Chapter 9. MPLINK Linker Overview

9.1 Introduction ............................................................................................... 171
9.2 MPLINK Linker Defined .............................................................................. 171
9.3 How MPLINK Linker Works ........................................................................ 171
9.4 How MPLINK Linker Helps You ................................................................. 172
9.5 Linker Platforms Supported ........................................................................ 172
9.6 Linker Operation ......................................................................................... 172
9.7 Linker Input/Output Files ............................................................................ 173

Chapter 10. Linker Interfaces

10.1 Introduction ............................................................................................... 179
10.2 MPLAB IDE Interface ............................................................................... 179
10.3 Command Line Interface .......................................................................... 179
10.4 Command Line Example .......................................................................... 180

Chapter 11. Linker Scripts

11.1 Introduction ............................................................................................... 181
11.2 Standard Linker Scripts ........................................................................... 181
11.3 Linker Script Command Line Information ................................................. 182
11.4 Linker Script Caveats ............................................................................... 183
11.5 Memory Region Definition ........................................................................ 183
11.6 Logical Section Definition ........................................................................ 186
11.7 STACK Definition ..................................................................................... 186
11.8 Conditional Linker Statements ................................................................. 187
<table>
<thead>
<tr>
<th>Chapter 12. Linker Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Introduction .......................................................... 191</td>
</tr>
<tr>
<td>12.2 Linker Processing Overview ........................................... 191</td>
</tr>
<tr>
<td>12.3 Linker Allocation Algorithm .......................................... 192</td>
</tr>
<tr>
<td>12.4 Relocation Example ..................................................... 193</td>
</tr>
<tr>
<td>12.5 Initialized Data ......................................................... 194</td>
</tr>
<tr>
<td>12.6 Reserved Section Names ............................................... 194</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 13. Sample Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Introduction ................................. 195</td>
</tr>
<tr>
<td>13.2 How to Build the Sample Applications ....................... 195</td>
</tr>
<tr>
<td>13.3 Sample Application 1 - Templates and Linker Scripts ........ 200</td>
</tr>
<tr>
<td>13.4 Sample Application 2 – Placing Code and Setting Config Bits ...... 203</td>
</tr>
<tr>
<td>13.5 Sample Application 3 – Using a Boot Loader .................. 206</td>
</tr>
<tr>
<td>13.6 Sample Application 4 – Configuring External Memory .......... 217</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 14. Errors, Warnings and Common Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Introduction .................................................. 223</td>
</tr>
<tr>
<td>14.2 Linker Parse Errors ......................................... 223</td>
</tr>
<tr>
<td>14.3 Linker Errors ................................................ 225</td>
</tr>
<tr>
<td>14.4 Linker Warnings ............................................. 230</td>
</tr>
<tr>
<td>14.5 COFF File Errors ........................................... 230</td>
</tr>
<tr>
<td>14.6 Other Errors, Warnings and Messages ................... 231</td>
</tr>
<tr>
<td>14.7 Common Problems ......................................... 231</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Part 3 – MPLIB Object Librarian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 15. MPLIB Librarian Overview</td>
</tr>
<tr>
<td>15.1 Introduction .................................................. 235</td>
</tr>
<tr>
<td>15.2 What is MPLIB Librarian ................................. 235</td>
</tr>
<tr>
<td>15.3 How MPLIB Librarian Works .............................. 235</td>
</tr>
<tr>
<td>15.4 How MPLIB Librarian Helps You .......................... 236</td>
</tr>
<tr>
<td>15.5 Librarian Operation ......................................... 236</td>
</tr>
<tr>
<td>15.6 Librarian Input/Output Files ............................. 237</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 16. Librarian Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1 Introduction ................................. 239</td>
</tr>
<tr>
<td>16.2 MPLAB IDE Interface ......................... 239</td>
</tr>
<tr>
<td>16.3 Command Line Options ......................... 240</td>
</tr>
<tr>
<td>16.4 Command Line Examples and Tips .............. 240</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 17. Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.1 Introduction .................................................. 241</td>
</tr>
<tr>
<td>17.2 Librarian Parse Errors ................................. 241</td>
</tr>
<tr>
<td>17.3 Library File Errors ......................................... 241</td>
</tr>
<tr>
<td>17.4 COFF File Errors ........................................... 242</td>
</tr>
</tbody>
</table>
### Part 4 – Utilities

#### Chapter 18. Utilities Overview and Usage

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 Introduction</td>
<td>245</td>
</tr>
<tr>
<td>18.2 What are Utilities</td>
<td>245</td>
</tr>
<tr>
<td>18.3 Utilities Operation</td>
<td>245</td>
</tr>
<tr>
<td>18.4 mp2hex.exe Utility</td>
<td>246</td>
</tr>
<tr>
<td>18.5 mp2cod.exe Utility</td>
<td>246</td>
</tr>
</tbody>
</table>

#### Chapter 19. Errors and Warnings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1 Introduction</td>
<td>247</td>
</tr>
<tr>
<td>19.2 Hex File Errors</td>
<td>247</td>
</tr>
<tr>
<td>19.3 COFF To COD Conversion Errors</td>
<td>247</td>
</tr>
<tr>
<td>19.4 COFF To COD Converter Warnings</td>
<td>247</td>
</tr>
<tr>
<td>19.5 COD File Errors</td>
<td>248</td>
</tr>
</tbody>
</table>

### Part 5 – Appendices

#### Appendix A. Instruction Sets

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Introduction</td>
<td>251</td>
</tr>
<tr>
<td>A.2 Key to 12/14-Bit Instruction Width Instruction Sets</td>
<td>251</td>
</tr>
<tr>
<td>A.3 12-Bit Instruction Width Instruction Set</td>
<td>253</td>
</tr>
<tr>
<td>A.4 14-Bit Instruction Width Instruction Set</td>
<td>254</td>
</tr>
<tr>
<td>A.5 14-Bit Instruction Width Extended Instruction Set</td>
<td>256</td>
</tr>
<tr>
<td>A.6 12-Bit/14-Bit Instruction Width Pseudo-Instructions</td>
<td>259</td>
</tr>
<tr>
<td>A.7 Key to PIC18 Device Instruction Set</td>
<td>260</td>
</tr>
<tr>
<td>A.8 PIC18 Device Instruction Set</td>
<td>261</td>
</tr>
<tr>
<td>A.9 PIC18 Device Extended Instruction Set</td>
<td>265</td>
</tr>
</tbody>
</table>

#### Appendix B. Useful Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Introduction</td>
<td>267</td>
</tr>
<tr>
<td>B.2 ASCII Character Set</td>
<td>267</td>
</tr>
<tr>
<td>B.3 Hexadecimal to Decimal Conversion</td>
<td>268</td>
</tr>
</tbody>
</table>

#### Glossary

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>269</td>
</tr>
</tbody>
</table>

#### Index

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>283</td>
</tr>
</tbody>
</table>

#### Worldwide Sales and Service

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>288</td>
</tr>
</tbody>
</table>
INTRODUCTION

This chapter contains general information that will be useful to know before using Assembler/Linker/Librarian User’s Guide. Items discussed include:

• Document Layout
• Conventions Used
• Recommended Reading
• The Microchip Web Site
• Development Systems Customer Change Notification Service
• Customer Support

DOCUMENT LAYOUT

This document describes how to use the MPASM™ assembler, the MPLINK™ object linker, and the MPLIB™ object librarian to develop code for PIC® microcontroller (MCU) applications. All of these tools can work within the MPLAB® Integrated Development Environment (IDE). For a detailed discussion about basic MPLAB IDE functions, refer to MPLAB IDE documentation.

PIC1X MCU Language Tools Overview – provides an overview of how to use all of the tools in this manual together under the MPLAB IDE. This is how most developers will use these tools.
Part 1 – “MPASM Assembler”

• Chapter 1. “MPASM Assembler Overview” – describes what the MPASM assembler is, what it does and how it works with other tools. Also, gives an overview of operation and discusses input/output files.

• Chapter 2. “Assembler Interfaces” – reviews how to use the MPASM assembler with MPLAB IDE and describes how to use the assembler on the command line or in a Windows shell interface.

• Chapter 3. “Expression Syntax and Operation” – provides guidelines for using complex expressions in MPASM assembler source files.

• Chapter 4. “Directives” – lists each MPASM assembler directive alphabetically and describes the directive in detail, with examples.

• Chapter 5. “Assembler Examples, Tips and Tricks” – provides examples of how to use the MPASM assembler directives together in applications.

• Chapter 6. “Relocatable Objects” – describes how to use the MPASM assembler in conjunction with MPLINK object linker.

• Chapter 7. “Macro Language” – describes how to use the MPASM assembler’s built-in macro processor.

• Chapter 8. “Errors, Warnings, Messages, and Limitations” – contains a descriptive list of the errors, warnings, and messages generated by the MPASM assembler, as well as tool limitations.

Part 2 – “MPLINK Object Linker”

• Chapter 9. “MPLINK Linker Overview” – describes what the MPLINK object linker is, what it does and how it works with other tools. Also, gives an overview of operation and discusses input/output files.

• Chapter 10. “Linker Interfaces” – reviews how to use the MPLINK linker with MPLAB IDE and describes how to use the linker on the command line.

• Chapter 11. “Linker Scripts” – discusses how to generate and use linker scripts to control linker operation.

• Chapter 12. “Linker Processing” – describes how the linker processes files.

• Chapter 13. “Sample Applications” – provides examples of how to use the linker to create applications.
  - Sample Application 1 – explains how to find and use template files and when to modify the generic linker script file.
  - Sample Application 2 – explains how to place program code in different memory regions, how to place data tables in ROM memory and how to set configuration bits in C.
  - Sample Application 3 – explains how to partition memory for a boot loader and how to compile code that will be loaded into external RAM and executed.
  - Sample Application 4 – explains how to create new linker script memory section, how to declare external memory through #pragma code directive, and how to access external memories using C pointers.

• Chapter 14. “Errors, Warnings and Common Problems” – contains a descriptive list of the errors and warnings generated by the MPLINK linker, as well as common problems and tool limitations.
Part 3 – “MPLIB Object Librarian”
• Chapter 15. “MPLIB Librarian Overview” – describes what the MPLIB object librarian is, what it does and how it works with other tools. Also, gives an overview of operation and discusses input/output files.
• Chapter 16. “Librarian Interfaces” – reviews how to use the MPLIB librarian with MPLAB IDE and describes how to use the librarian on the command line.
• Chapter 17. “Errors” – contains a descriptive list of the errors generated by the MPLIB librarian.

Part 4 – “Utilities”
• Chapter 18. “Utilities Overview and Usage” – lists the available utilities and describes their usage.
• Chapter 19. “Errors and Warnings” – contains a descriptive list of the errors generated by the utilities.

Part 5 – “Appendices”
• Appendix A. “Instruction Sets” – lists PIC MCU device instruction sets.
• Appendix B. “Useful Tables” – provides some useful tables for code development.
  - ASCII Character Set – lists the ASCII Character Set.
  - Hexadecimal to Decimal Conversions – shows how to convert from hexadecimal to decimal numbers.
CONVENTIONS USED

The following conventions may appear in this documentation:

### DOCUMENTATION CONVENTIONS

<table>
<thead>
<tr>
<th>Description</th>
<th>Represents</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arial font:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Italic characters</td>
<td>Referenced books</td>
<td><em>MPLAB IDE User’s Guide</em></td>
</tr>
<tr>
<td></td>
<td>Emphasized text</td>
<td><em>...is the only compiler...</em></td>
</tr>
<tr>
<td>Initial caps</td>
<td>A window</td>
<td>The Output window</td>
</tr>
<tr>
<td></td>
<td>A dialog</td>
<td>The Settings dialog</td>
</tr>
<tr>
<td></td>
<td>A menu selection</td>
<td>Select Enable Programmer</td>
</tr>
<tr>
<td>Quotes</td>
<td>A field name in a window or dialog</td>
<td><em>“Save project before build”</em></td>
</tr>
<tr>
<td>Underlined, italic text with</td>
<td>A menu path</td>
<td><em>File&gt;Save</em></td>
</tr>
<tr>
<td>right angle bracket</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bold characters</td>
<td>A dialog button</td>
<td>Click <strong>OK</strong></td>
</tr>
<tr>
<td></td>
<td>A tab</td>
<td>Click the <strong>Power</strong> tab</td>
</tr>
<tr>
<td>Text in angle brackets &lt; &gt;</td>
<td>A key on the keyboard</td>
<td>Press <strong>&lt;Enter&gt;, &lt;F1&gt;</strong></td>
</tr>
<tr>
<td><strong>Courier font:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plain Courier</td>
<td>Sample source code</td>
<td><em>#define START</em></td>
</tr>
<tr>
<td></td>
<td>Filenames</td>
<td><em>autoexec.bat</em></td>
</tr>
<tr>
<td></td>
<td>File paths</td>
<td><em>c:\mcc18\h</em></td>
</tr>
<tr>
<td></td>
<td>Keywords</td>
<td><em>_asm, _endasm, static</em></td>
</tr>
<tr>
<td></td>
<td>Command-line options</td>
<td><em>-Opa+, -Opa-</em></td>
</tr>
<tr>
<td></td>
<td>Bit values</td>
<td><em>0, 1</em></td>
</tr>
<tr>
<td></td>
<td>Constants</td>
<td><em>0xFF, ‘A’</em></td>
</tr>
<tr>
<td>Italic Courier</td>
<td>A variable argument</td>
<td><em>file.o, where file can be any valid filename</em></td>
</tr>
<tr>
<td>Square brackets []</td>
<td>Optional arguments</td>
<td><em>mpasmwin [options]</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>file [options]</em></td>
</tr>
<tr>
<td>Curly brackets and pipe</td>
<td>Choice of mutually exclusive arguments; an OR selection</td>
<td>*errorlevel {0</td>
</tr>
<tr>
<td>character: {</td>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Ellipses...</td>
<td>Replaces repeated text</td>
<td><em>var_name [, var_name...]</em></td>
</tr>
<tr>
<td></td>
<td>Represents code supplied by user</td>
<td>*void main (void)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*{ ...}</td>
</tr>
</tbody>
</table>
RECOMMENDED READING

This documentation describes how to use Assembler/Linker/Librarian User’s Guide. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

**Readme Files - readme.asm and readme.lkr**

For the latest tool information and known issues, see the MPASM assembler readme file (Readme for MPASM Assembler.htm) or the MPLINK object linker/MPLIB object librarian readme file (Readme for MPLINK Linker.htm). These ASCII text files may be found in the Readme folder of the MPLAB IDE installation directory.

**On-line Help Files**

Comprehensive help files are available for MPASM assembler and MPLINK object linker/MPLIB object librarian. In addition, debug output format (COFF) information is also available in help.

**C Compiler User’s Guides and Libraries**

The MPLINK linker and MPLIB librarian also work with the MPLAB C Compiler for PIC18 MCUs (formerly MPLAB C18). For more information on the compiler, see:

- MPLAB® C Compiler for PIC18 MCUs Getting Started (DS51295)
- MPLAB® C Compiler for PIC18 MCUs User’s Guide (DS51288)
- MPLAB® C Compiler for PIC18 MCUs Libraries (DS51297)

**MPLAB IDE Documentation**

Information on the integrated development environment MPLAB IDE may be found in:

- MPLAB® IDE User’s Guide (DS51519) – Comprehensive user’s guide
- On-line help file – The most up-to-date information on MPLAB IDE

**PIC MCU Data Sheets and Application Notes**

Data sheets contain information on device operation, as well as electrical specifications. Applications notes demonstrate how various PIC MCU’s may be used. Find both of these types of documents for your device on the Microchip website.
THE MICROCHIP WEB SITE

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

• **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user’s guides and hardware support documents, latest software releases and archived software

• **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip consultant program member listing

• **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

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

The Development Systems product group categories are:

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

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

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

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

• **Programmers** – The latest information on Microchip programmers. These include the MPLAB PM3 and PRO MATE II device programmers and the PICSTART® Plus and PICkit™ 1 development programmers.
CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

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

The MPASM assembler, the MPLINK object linker and the MPLIB object librarian are typically used together under MPLAB IDE to provide GUI development of application code for PIC1X MCU devices (PIC10/12/16/18 MCUs). The operation of these 8-bit language tools with MPLAB IDE is discussed here.

Topics covered in this chapter:
- MPLAB IDE and Tools Installation
- MPLAB IDE Setup
- MPLAB IDE Projects
- Project Setup
- Project Example

MPLAB IDE AND TOOLS INSTALLATION

In order to use the PIC® MCU language tools with MPLAB IDE, you must first install MPLAB IDE. The latest version of this free software is available at our website (http://www.microchip.com) or from any sales office (back cover). When you install MPLAB IDE, you will be installing the MPASM assembler, the MPLINK object linker and the MPLIB object librarian as well.

The language tools will be installed, by default, in the directory:
- C:\Program Files\Microchip\MPASM Suite

The executables for each tool will be:
- MPASM Assembler - mpasmwin.exe
- MPLINK Object Linker - mplink.exe
- MPLIB Object Librarian - mplib.exe
- Other Utilities

All device include (header) files are also in this directory. For more on these files, see MPASM assembler documentation.

All device linker script files are in the LKR subdirectory. For more on these files, see MPLINK object linker documentation.

Code examples and template files are also included in subdirectories for your use. Template files are provided for absolute code (Code) and relocatable code (Object) development.
MPLAB IDE SETUP

Once MPLAB IDE is installed on your PC, check the settings below to ensure that the language tools are properly recognized under MPLAB IDE.

1. From the MPLAB IDE menu bar, select Project>Set Language Tool Locations to open a dialog to set/check language tool executable location.

2. In the dialog, under “Registered Tools”, select “Microchip MPASM Toolsuite”. Click the “+” to expand.
3. Select “Executables”. Click the “+” to expand.
4. Select “MPASM Assembler (mpasmwin.exe)”. Under “Location”, a path to the executable file should be displayed. If no path is displayed, enter one or browse to the location of this file. The default location is listed in “MPLAB IDE and Tools Installation”.
5. Select “MPLINK Object Linker (mplink.exe)”. Under “Location”, a path to the executable file should be displayed. If no path is displayed, enter one or browse to the location of this file. The default location is listed in “MPLAB IDE and Tools Installation”.
6. Select “MPLIB Object Librarian (mplib.exe)”. Under “Location”, a path to the executable file should be displayed. If no path is displayed, enter one or browse to the location of this file. The default location is listed in “MPLAB IDE and Tools Installation”.
7. Click OK.
MPLAB IDE PROJECTS

A project in MPLAB IDE is a group of files needed to build an application, along with their associations to various build tools. Below is a generic MPLAB IDE project.

FIGURE 2: PROJECT RELATIONSHIPS

* The linker can select this file for you.
In this MPLAB IDE project, an assembly source file (`prog.asm`) is shown with its associated assembler (MPASM assembler). MPLAB IDE will use this information to generate the object file `prog.o` for input into the MPLINK object linker. For more information on the assembler, see the MPASM assembler documentation.

The C source file `main.c` is also shown with its associated MPLAB C18 C compiler. MPLAB IDE will use this information to generate an object file (`main.o`) for input into the MPLINK object linker. For more information on the compiler, see the MPLAB C18 C compiler documentation listed in Recommended Reading.

In addition, precompiled object files (`precomp.o`) may be included in a project, with no associated tool required. MPLAB C18 requires the inclusion of a precompiled standard code module `c018i.o`, for example. For more information on available Microchip precompiled object files, see the MPLAB C18 C compiler documentation.

Some library files (`math.lib`) are available with the compiler. Others may be built using the librarian tool (MPLIB object librarian). For more information on the librarian, see the MPLIB librarian documentation. For more information on available Microchip libraries, see the MPLAB C18 C compiler documentation.

The object files, along with library files, are used to generate the project output files via the linker (MPLINK object linker). Depending on your project, you may or may not need to add a linker script file (`device.lkr`). For more information on using linker script files and the linker, see the MPLINK linker documentation.

The main output file generated by the MPLINK linker is the COF file (`prog.cof`). The linker then uses the utility MP2HEX to generate the Hex file (`prog.hex`), used by simulators, emulators, debuggers and programmers. For more information on linker output files, see the MPLINK linker documentation. For more information on utilities, see the related documentation.

For more on projects, and related workspaces, see MPLAB IDE documentation.
PROJECT SETUP

To set up an MPLAB IDE project for the first time, it is advisable to use the built-in Project Wizard (Project>Project Wizard). In this wizard, you will be able to select a language toolsuite that uses MPASM assembler, e.g., the Microchip MPASM Toolsuite. For more on the wizard, and MPLAB IDE projects, see MPLAB IDE documentation.

Once you have a project set up, you may then set up properties of the tools in MPLAB IDE.

1. From the MPLAB IDE menu bar, select Project>Build Options>Project to open a dialog to set/check project build options.

   ![Note: MPASM assembler does not recognize include path information specified in MPLAB IDE.]

2. Click on the tool tab to modify tool settings.
   - Build Options Dialog, MPASM Assembler Tab
   - Build Options Dialog, MPLAB C17 Tab (If Installed)
   - Build Options Dialog, MPLAB C18 Tab (If Installed)
   - Build Options Dialog, MPLINK Linker Tab
   - Build Options Dialog, MPASM/C17/C18 Suite Tab

**Build Options Dialog, MPASM Assembler Tab**

Select a category, and then set up assembler options. For additional options, see MPASM assembler documentation, Chapter 2. “Assembler Interfaces”.

**General Category**

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
</table>
| Disable case sensitivity | The assembler will not distinguish between upper- and lower-case letters.  
**Note:** Disabling case sensitivity will make all labels uppercase. |
| Extended mode | Enable PIC18F extended instruction support. |
| Default Radix | Set the default radix, either Hexadecimal, Decimal or Octal. |
| Macro Definitions | Add macro directive definitions. |
| Restore Defaults | Restore tab default settings. |

**Use Alternate Settings**

| Text Box | Enter options in a command-line (non-GUI) format. |

**Output Category**

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostics level</td>
</tr>
<tr>
<td>Generate cross-reference file</td>
</tr>
<tr>
<td>Hex file format (for single-file assemblies)</td>
</tr>
<tr>
<td>Restore Defaults</td>
</tr>
</tbody>
</table>

**Use Alternate Settings**

| Text Box | Enter options in a command-line (non-GUI) format. |
Build Options Dialog, MPLAB C17 Tab (If Installed)

Although the MPLAB C17 C compiler works with MPLAB IDE, it must be acquired separately. See the Microchip website (www.microchip.com) for details. This compiler supports PIC17C MCU devices.

Note: PIC17C MCUs are end-of-life devices. Consider migrating to PIC18X MCU devices.

A subset of command-line options may be specified in MPLAB IDE in the Build Options dialog, MPLAB C17 tab. Select a category, and then set up compiler options. For additional options, see the MPLAB C17 C Compiler User’s Guide (DS51290), also available on the Microchip website.

General Category

Generate Command Line

Diagnostics level
Select to display errors only; errors and warnings; or errors, warnings and messages. These will be shown in the Output window.

Default storage class
Select the storage class, either ANSI-standard auto or static.

Macro Definitions
Add macro directive definitions.

Restore Defaults
Restore tab default settings.

Use Alternate Settings

Text Box
Enter options in a command-line (non-GUI) format.

Memory Model Category

Generate Command Line

Small
near rom – program memory ≤ 8K, near ram – data memory ≤ 256

Medium
far rom – program memory > 8K, near ram – data memory ≤ 256

Compact
near rom – program memory ≤ 8K, far ram – data memory > 256

Large
far rom – program memory > 8K, far ram – data memory > 256

Restore Defaults
Restore tab default settings.

Use Alternate Settings

Text Box
Enter options in a command-line (non-GUI) format.

Optimization Category

Generate Command Line

Bank Selection Optimization
Select the level of bank selection optimization. Removes MOVLB instruction in instances where it can be determined that the Bank Select register already contains the correct value.

Level 0 – None.
Level 1 – Equivalent to -On1.
Level 2 – Equivalent to -On2.

Other Optimizations
Select individual optimizations.

Restore Defaults
Restore tab default settings.

Use Alternate Settings

Text Box
Enter options in a command-line (non-GUI) format.
Build Options Dialog, MPLAB C18 Tab (If Installed)

Although the MPLAB C18 C compiler works with MPLAB IDE, it must be acquired separately. The full version may be purchased, or a student (limited-feature) version may be downloaded for free. See the Microchip website (www.microchip.com) for details. This compiler supports PIC18X MCU devices.

A subset of command-line options may be specified in MPLAB IDE in the Build Options dialog, MPLAB C18 tab. Select a category, and then set up compiler options. For additional options, see the MPLAB C18 C Compiler User’s Guide (DS51288), also available on the Microchip website.

### General Category

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostics level</td>
</tr>
<tr>
<td>Default storage class</td>
</tr>
<tr>
<td>Enable integer promotions</td>
</tr>
<tr>
<td>Treat ‘char’ as unsigned</td>
</tr>
<tr>
<td>Extended mode</td>
</tr>
<tr>
<td>Macro Definitions</td>
</tr>
<tr>
<td>Restore Defaults</td>
</tr>
<tr>
<td><strong>Use Alternate Settings</strong></td>
</tr>
</tbody>
</table>

### Memory Model Category

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Model</td>
</tr>
<tr>
<td>Data Model</td>
</tr>
<tr>
<td>Stack Model</td>
</tr>
<tr>
<td>Restore Defaults</td>
</tr>
<tr>
<td><strong>Use Alternate Settings</strong></td>
</tr>
</tbody>
</table>

### Optimization Category

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable</td>
</tr>
<tr>
<td>Debug</td>
</tr>
<tr>
<td>Enable All</td>
</tr>
<tr>
<td>Custom</td>
</tr>
<tr>
<td>Procedural Abstraction passes</td>
</tr>
<tr>
<td>Restore Defaults</td>
</tr>
<tr>
<td><strong>Use Alternate Settings</strong></td>
</tr>
</tbody>
</table>
Build Options Dialog, MPLINK Linker Tab

Select a category, and then set up linker options. For additional options, see MPLINK object linker documentation, Chapter 10. “Linker Interfaces”.

All Options Category

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex file format</td>
</tr>
<tr>
<td>Generate map file</td>
</tr>
<tr>
<td>Output file root</td>
</tr>
<tr>
<td>Restore Defaults</td>
</tr>
</tbody>
</table>

Use Alternate Settings

Text Box

Enter options in a command-line (non-GUI) format.

Build Options Dialog, MPASM/C17/C18 Suite Tab

Select a category, and then set up output build options.

All Options Category

<table>
<thead>
<tr>
<th>Generate Command Line</th>
</tr>
</thead>
</table>
| Build normal target  | The files in the project will be built for normal output using the MPLINK linker (hex file, etc.)
| (invoke MPLINK)       | To set linker options, see Build Options Dialog, MPLINK Linker Tab. |
| Build library target | The files in the project will be built into a library using the MPLIB librarian (lib file.)
| (invoke MPLIB)        | For a library build, a generic device-family library (for the selected device) may be built, e.g., for the selected device PIC18F8720, the generic device-family library would be PIC18. For more on libraries, see MPLIB object librarian documentation, Chapter 16. “Librarian Interfaces”. |
PROJECT EXAMPLE

In this example, you will create an MPLAB IDE project with multiple assembly files. Therefore, you will need to use the MPASM assembler and the MPLINK linker to create the final output executable (.hex) file.

- Run the Project Wizard
- Set Build Options
- Build the Project
- Build Errors
- Output Files
- Further Development

Run the Project Wizard

In MPLAB IDE, select Project>Project Wizard to launch the wizard. Click Next> at the Welcome screen.

1. Select PIC16F84A as the Device. Click Next> to continue.
2. Set up the language tools, if you haven’t already. Refer to “MPLAB IDE Setup”. Click Next> to continue.
3. Enter “Example” for the name of the project. Then Browse to select a location for your project. Click Next> to continue.
4. Add files to the project. In the file listing box on the left of the dialog, find the following directory:
   C:\Program Files\Microchip\MPASM Suite\EXAMPLE
   Select Example.asm and Example2.asm. Click Add>> to add these files to the project. Click Next> to continue.
5. Review the summary of information. If anything is in error, use <Back to go back and correct the entry. Click Finish to complete the project creation and setup.

Once the Project Wizard has completed, the Project window should contain the project tree. The workspace name is Example.mcw, the project name is Example.mcp, and all the project files are listed under their respective file type. For more on workspaces and projects, see MPLAB IDE documentation.

FIGURE 3: EXAMPLE PROJECT TREE
Set Build Options

Select **Project>Build Options>Project** to open the Build Options dialog.

1. Click on the **MPASM Assembler** tab. For “Categories: General”, check that the “Default Radix” is set to “Hexadecimal”. For “Categories: Output”, check that the “Diagnostics level” includes all errors, warnings and messages. Then check the checkbox for “Generate cross-reference file”.
2. Click on the **MPLINK Linker** tab. For “Categories: (All Options)”, check that the “Hex File Format” is set to “INHX32”. Then check the checkbox for “Generate map file”.
3. Click on the **MPASM/C17/C18 Suite** tab. For “Categories: (All Options)”, check that the “Build normal target (invoke MPLINK)” is selected.
4. Click **OK** on the bottom of the dialog to accept the build options and close the dialog.
5. Select **Project>Save Project** to save the current configuration of the Example project.

Build the Project

Select **Project>Build All** to build the project.

**Note:** You also may right-click on the project name, “Example.mcp”, in the project tree and select “Build All” from the pop-up menu.

The Output window should appear at the end of the build and display the build results.

**FIGURE 4:** OUTPUT WINDOW – BUILD TAB

Build Errors

If the build did not complete successfully, check these items:

1. Review the previous steps in this example. Make sure you have set up the language tools correctly and have all the correct project files and build options.
2. If you modified the sample source code, examine the Build tab of the Output window for syntax errors in the source code. If you find any, double-click on the error to go to the source code line that contains that error. Correct the error, and then try to build again.
Output Files

View the project output files by opening the files in MPLAB IDE.

1. Select **File>Open**. In the Open dialog, find the project directory.
2. Under “Files of type” select “All files (*.*)” to see all project files.
3. Select “Example.xrf”. Click **Open** to view the assembler cross-reference file for Example.asm in an MPLAB IDE editor window. For more on this file, see the MPASM assembler documentation, Section 1.7.6 “Cross Reference File (.xrf)”.
4. Repeat steps 1 and 2. Select “Example.map”. Click **Open** to view the linker map file in an MPLAB IDE editor window. For more on this file, see the MPLINK linker documentation, Section 9.7.7 “Map File (.map)”.
5. Repeat steps 1 and 2. Select “Example.lst”. Click **Open** to view the linker listing file in an MPLAB IDE editor window. When MPASM assembler is used with MPLINK linker, the listing file is generated by the linker. For more on this file, see the MPLINK linker documentation, Section 9.7.6 “Listing File (.lst)”.
6. Repeat steps 1 and 2. Notice that there is only one hex file, “Example.hex”. This is the primary output file, used by various debug tools. You do not view this file for debugging; use instead **View>Program Memory** or **View>Disassembly Listing**.

Further Development

Usually, your application code will contain errors and not build the first time. Therefore, you will need a debug tool to help you develop your code. Using the output files previously discussed, several debug tools exist that work with MPLAB IDE to help you do this. You may choose from simulators, in-circuit emulators or in-circuit debuggers, either manufactured by Microchip Technology or third-party developers. Please see the documentation for these tools to see how they can help you. When debugging, you will need to set the Build Configuration to “Debug”. Please see MPLAB IDE documentation concerning this control.

Once you have developed your code, you will want to program it into a device. Again, there are several programmers that work with MPLAB IDE to help you do this. Please see the documentation for these tools to see how they can help you. When programming, you will need to set the Build Configuration to “Release”. Please see MPLAB IDE documentation concerning this control.

For more information on using MPLAB IDE, consult the on-line help that comes with this application or download printable documents from our website.
Chapter 1. MPASM Assembler Overview

1.1 INTRODUCTION

An overview of the MPASM assembler and its capabilities is presented. Topics covered in this chapter:

• MPASM Assembler Defined
• How MPASM Assembler Helps You
• Assembler Migration Path
• Assembler Compatibility Issues
• Assembler Operation
• Assembler Input/Output Files

1.2 MPASM ASSEMBLER DEFINED

The MPASM assembler (the assembler) is a command-line or Windows-based PC application that provides a platform for developing assembly language code for Microchip's PIC1X microcontroller (MCU) families.

The executable version of the assembler is `mpasmwin.exe`. Use this version with MPLAB IDE, in a stand-alone Windows application, or on the command line. This version is available with MPLAB IDE or with the regular and demo version of the MPLAB C18 C compiler.

The MPASM assembler supports all PIC1X MCU devices, as well as memory and KeeLoq® secure data products from Microchip Technology Inc. (Some memory and KeeLoq devices were not supported in MPLAB IDE after v5.70.40.)

1.3 HOW MPASM ASSEMBLER HELPS YOU

The MPASM assembler provides a universal solution for developing assembly code for all of Microchip's PIC1X MCUs. Notable features include:

• MPLAB IDE Compatibility
• Windows/Command Line Interfaces
• Rich Directive Language
• Flexible Macro Language

1.4 ASSEMBLER MIGRATION PATH

Since the MPASM assembler is a universal assembler for all PIC1X MCU devices, application code developed for the PIC16F877A can be translated into a program for the PIC18F452. This may require changing the instruction mnemonics that are not the same between the devices (assuming that register and peripheral usage were similar). Also, configuration settings may be different. The __CONFIG syntax with one operand is not recognized by PIC18 and PIC16F19XX MCUs. The rest of the directive and macro language will be the same.
1.5 ASSEMBLER COMPATIBILITY ISSUES

The MPASM assembler is compatible with the MPLAB IDE integrated development environment and all Microchip PIC1X MCU development systems currently in production.

The MPASM assembler supports a clean and consistent method of specifying radix (see Section 3.4 “Numeric Constants and Radix”). You are encouraged to develop using the radix and other directive methods described within this document, even though certain older syntaxes may be supported for compatibility reasons.

1.6 ASSEMBLER OPERATION

The MPASM assembler can be used in two ways:

• To generate absolute code that can be executed directly by a microcontroller.
• To generate relocatable code that can be linked with other separately assembled or compiled modules.

1.6.1 Generating Absolute Code

Absolute code is the default output from the MPASM assembler. This process is shown below.

When a source file is assembled in this manner, all variables and routines used in the source file must be defined within that source file, or in files that have been explicitly included by that source file. If assembly proceeds without errors, a hex file will be generated, containing the executable machine code for the target device. This file can then be used with a debugger to test code execution or with a device programmer to program the microcontroller.
1.6.2 Generating Relocatable Code

The MPASM assembler also has the ability to generate a relocatable object module that can be linked with other modules using Microchip's MPLINK linker to form the final executable code. This method is very useful for creating reusable modules.

Related modules can be grouped and stored together in a library using Microchip's MPLIB librarian. Required libraries can be specified at link time, and only the routines that are needed will be included in the final executable.

Refer to Chapter 6. “Relocatable Objects” for more information on the differences between absolute and relocatable object assembly.
1.7  ASSEMBLER INPUT/OUTPUT FILES

These are the default file extensions used by the assembler and the associated utility functions.

<table>
<thead>
<tr>
<th>TABLE 1-1: INPUT FILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Code (.asm)</td>
</tr>
<tr>
<td>Include File (.inc)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 1-2: OUTPUT FILES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listing File (.lst)</td>
</tr>
<tr>
<td>Error File (.err)</td>
</tr>
<tr>
<td>Hex File Formats (.hex, .hxl, .hxh)</td>
</tr>
<tr>
<td>Cross Reference File (.xrf)</td>
</tr>
<tr>
<td>Object File (.o)</td>
</tr>
</tbody>
</table>

1.7.1  Source Code (.asm)

Assembly is a programming language you may use to develop the source code for your application. The source code file may be created using any ASCII text file editor.

Note: Several example source code files are included free with MPLAB IDE.

Your source code should conform to the following basic guidelines.

Each line of the source file may contain up to four types of information:

- Labels
- Mnemonics, Directives and Macros
- Operands
- Comments

The order and position of these are important. For ease of debugging, it is recommended that labels start in column one and mnemonics start in column two or beyond. Operands follow the mnemonic. Comments may follow the operands, mnemonics or labels, and can start in any column. The maximum column width is 255 characters.

White space or a colon must separate the label and the mnemonic, and white space must separate the mnemonic and the operand(s). Multiple operands must be separated by commas.

White space is one or more spaces or tabs. White space is used to separate pieces of a source line. White space should be used to make your code easier for people to read. Unless within character constants, any white space means the same as exactly one space.
EXAMPLE 1-1: ABSOLUTE MPASM ASSEMBLER SOURCE CODE (SHOWS MULTIPLE OPERANDS)

```
list      p=18f452
#include  p18f452.inc

Dest  equ       0x0B    ;Define constant
org       0x0000  ;Reset vector
goto      Start
org       0x0020  ;Begin program

Start
  movlw     0x0A
  movwf     Dest
  bcf       Dest, 3 ;This line uses 2 operands
  goto      Start
end
```

1.7.1.1 LABELS

A label is used to represent a line or group of code, or a constant value. It is needed for branching instructions (Example 1-1.)

Labels should start in column 1. They may be followed by a colon (:), space, tab or the end of line. Labels must begin with an alpha character or an under bar (_) and may contain alphanumeric characters, the under bar and the question mark.

Labels must not:

• begin with two leading underscores, e.g., __config.
• begin with a leading underscore and number, e.g., _2NDLOOP.
• be an assembler reserved word (see Section 3.3 “Reserved Words and Section Names”).

Labels may be up to 32 characters long. By default they are case sensitive, but case sensitivity may be overridden by a command-line option (/c). If a colon is used when defining a label, it is treated as a label operator and not part of the label itself.

1.7.1.2 MNEMONICS, DIRECTIVES AND MACROS

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

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

Macros are user defined sets of instructions and directives that will be evaluated in-line with the assembler source code whenever the macro is invoked.
Assembler instruction mnemonics, directives and macro calls should begin in column two or greater. If there is a label on the same line, instructions must be separated from that label by a colon, or by one or more spaces or tabs.

1.7.1.3 OPERANDS
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. Multiple operands must be separated by commas.

1.7.1.4 COMMENTS
Comments are text explaining the operation of a line or lines of code.
The MPASM assembler treats anything after a semicolon as a comment. All characters following the semicolon are ignored through the end of the line. String constants containing a semicolon are allowed and are not confused with comments.

1.7.2 Include File (.inc)
An assembler include, or header, file is any file containing valid assembly code. Usually, the file contains device-specific register and bit assignments. This file may be “included” in the code so that it may be reused by many programs.
As an example, to add the standard header file for the PIC18F452 device to your assembly code, use:
#include p18f452.inc
Standard header files are located in:
C:\Program Files\Microchip\MPASM Suite

1.7.3 Listing File (.lst)
An MPASM assembler listing file provides a mapping of source code to object code. It also provides a list of symbol values, memory usage information, and the number of errors, warnings and messages generated. This file may be viewed in MPLAB IDE by:
1. selecting File>Open to launch the Open dialog
2. selecting “List files (*.lst)” from the “Files of type” drop-down list
3. locating the desired list file
4. clicking on the list file name
5. clicking Open
Both the MPASM assembler and the MPLINK linker can generate listing files. For information on the MPLINK linker listing file, see 9.7.6 “Listing File (.lst)”.
To prevent assembler list file generation, use the /l- option or use with MPLINK linker (The linker list file overwrites the assembler list file.) Set the size of tabs in the list file using the /t option.

EXAMPLE 1-2: ABSOLUTE MPASM ASSEMBLER LISTING FILE
The product name and version, the assembly date and time, and the page number appear at the top of every page.
The first column contains the base address in memory where the code will be placed. The second column displays the 32-bit value of any symbols created with the set, equ, variable, constant, or cblob directive. The third column is reserved for the machine instruction. This is the code that will be executed by the PIC1X MCU. The fourth column lists the associated source file line number for this line. The remainder of the line is reserved for the source code line that generated the machine code.
Errors, warnings, and messages are embedded between the source lines and pertain to the following source line. Also, there is a summary at the end of the listing.

The symbol table lists all symbols defined in the program.

The memory usage map gives a graphical representation of memory usage. 'X' marks a used location and '-' marks memory that is not used by this object. The map also displays program memory usage. The memory map is not printed if an object file is generated.

**Note:** Due to page width restrictions, some comments have been shortened, indicated by "..". Also, some symbol table listings have been removed, indicated by ":". See the standard header, p18f452.inc, for a complete list of symbols.
1.7.4  Error File (.err)

The MPASM assembler, by default, generates an error file. This file can be useful when debugging your code. The MPLAB IDE will display the error information in the Output window. The format of the messages in the error file is:

```
type[number] file line description
```

For example:

```
Error[113] C:\PROG.ASM 7 : Symbol not previously defined (start)
```

The error file may contain any number of MPASM assembler errors, warnings and messages. For more on these, see Chapter 8. “Errors, Warnings, Messages, and Limitations”.

To prevent error file generation, use the /e- option.

1.7.5  Hex File Formats (.hex, .hxl, .hxh)

The MPASM assembler and MPLINK linker are capable of producing ASCII text hex files in different formats.

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Format Type</th>
<th>File Extension</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Hex Format</td>
<td>INHX8M</td>
<td>.hex</td>
<td>8-bit core device programmers</td>
</tr>
<tr>
<td>Intel Split Hex Format</td>
<td>INHX8S</td>
<td>.hxl, .hxh</td>
<td>odd/even programmers</td>
</tr>
<tr>
<td>Intel Hex 32 Format</td>
<td>INHX32</td>
<td>.hex</td>
<td>16-bit core device programmers</td>
</tr>
</tbody>
</table>

This file format is useful for transferring PIC1X MCU series code to Microchip programmers and third party PIC1X MCU programmers.

1.7.5.1  INTEL HEX FORMAT

This format produces one 8-bit hex file with a low byte, high byte combination. Since each address can only contain 8 bits in this format, all addresses are doubled.

Each data record begins with a 9-character prefix and ends with a 2-character checksum. Each record has the following format:

```
:BBAAAAATTHHHH.....HHHCC
```

where:

- **BB**  A two digit hexadecimal byte count representing the number of data bytes that will appear on the line.
- **AAAA**  A four digit hexadecimal address representing the starting address of the data record.
- **TT**  A two digit record type that will always be '00' except for the end-of-file record, which will be '01'.
- **HH**  A two digit hexadecimal data byte, presented in low byte/high byte combinations.
- **CC**  A two digit hexadecimal checksum that is the two's complement of the sum of all preceding bytes in the record.
EXAMPLE 1-3: INHX8M

```plaintext
file_name.hex
:1000000000000000000000000000000000000000F0
:040010000000000EC
:10002006801A9018901EA01280208026A02BF02C5
:10005200E0228036803BF03E803C8030804B8
:1000620008040804030443050306E807E807FF0839
:06007200FF08FF08190A57
:00000001FF
```

1.7.5.2 INTEL SPLIT HEX FORMAT

The split 8-bit file format produces two output files: `.hxl` and `.hxh`. The format is the same as the normal 8-bit format, except that the low bytes of the data word are stored in the `.hxl` file, and the high bytes of the data word are stored in the `.hxh` file, and the addresses are divided by two. This is used to program 16-bit words into pairs of 8-bit EPROMs, one file for low byte, one file for high byte.

EXAMPLE 1-4: INHX8S

```plaintext
file_name.hxl
:0A0000000000000000000000000000F6
:1000190000284068A8E8C82868A989EA28086ABFAA
:10002900E0E82868BFE8C8080803430303E8E8FFD0
:03003900FFFF19AD
:00000001FF

file_name.hxh
:0A0000000000000000000000000000F6
:1000190000000000000000010101010102020202CA
:10002900020203030303030404040405050607070883
:0300390008080AAA
:00000001FF
```

1.7.5.3 INTEL HEX 32 FORMAT

The extended 32-bit address hex format is similar to the hex 8 format, except that the extended linear address record is output also to establish the upper 16 bits of the data address. This is mainly used for 16-bit core devices since their addressable program memory exceeds 64 kbytes.

Each data record begins with a 9-character prefix and ends with a 2-character checksum. Each record has the following format:

```
:BBAAAATTHHHH....HHHCC
```

where:

- **BB** A two digit hexadecimal byte count representing the number of data bytes that will appear on the line.
- **AAAA** A four digit hexadecimal address representing the starting address of the data record.
- **TT** A two digit record type:
  - 00 - Data record
  - 01 - End of File record
  - 02 - Segment address record
  - 04 - Linear address record
- **HH** A two digit hexadecimal data byte, presented in low byte/high byte combinations.
- **CC** A two digit hexadecimal checksum that is the two's complement of the sum of all preceding bytes in the record.
1.7.6 Cross Reference File (.xrf)

A cross reference file contains a listing of all symbols used in the assembly code. The file has the following format:

- The symbols are listed in the “Label” column, sorted by name.
- The “Type” column defines the type of symbol. A list of “Label Types” is provided at the end of the file.
- The “File Name” column lists the names of the files that use the symbol.
- The “Source File References” column lists the line number of the corresponding file in the “File Name” column where the symbol is defined/referenced. An asterisk means a definition.

To prevent cross-reference file generation, use the /x- option.

1.7.7 Object File (.o)

The assembler creates a relocatable object file from source code. This object file does not yet have addresses resolved and must be linked before it can be used as an executable.

To generate a file that will execute after being programmed into a device, see 1.7.5 “Hex File Formats (.hex, .hxl, .hxh)".

To prevent object file generation, use the /o- option.
Chapter 2. Assembler Interfaces

2.1 INTRODUCTION

There are several interfaces with which you may use the MPASM assembler, depending on the assembler version. These interfaces are discussed here.

When MPLAB IDE is installed, the MPASM assembler (mpasmwin.exe) is also installed. In addition, the assembler may be obtained with the regular and demo version of the MPLAB C18 C compiler.

Topics covered in this chapter:
- MPLAB IDE Interface
- Windows Interface
- Command Line Interface

2.2 MPLAB IDE INTERFACE

The MPASM assembler is most commonly used with the MPLINK linker in an MPLAB IDE project to generate relocatable code. For more information on this use, see “PIC1X MCU Language Tools and MPLAB IDE”.

The assembler may also be used in MPLAB IDE to generate absolute code (without the use of the MPLINK linker or MPLAB IDE project) by using the QuickBuild feature. To do this:

1. From the MPLAB IDE menu bar, select Project>Set Language Tool Locations to open a dialog to set/check language tool executable location.
2. In the dialog, under Registered Tools, select "Microchip MPASM Toolsuite". Click the “+” to expand.
3. Select Executables. Click the “+” to expand.
4. Select MPASM Assembler (mpasmwin.exe). Under Location, a path to the mpasmwin.exe file should be displayed. If no path is displayed, enter one or browse to the location of this file. By default, it is located at:
   C:\Program Files\Microchip\MPASM Suite\mpasmwin.exe
5. Click OK.
6. From the MPLAB IDE menu bar, select Project>Quickbuild to assemble the specified asm file using the MPASM assembler.
2.3 WINDOWS INTERFACE

MPASM assembler for Windows provides a graphical interface for setting assembler options. It is invoked by executing \texttt{mpasmwin.exe} in Windows Explorer or from a command prompt.

\textbf{FIGURE 2-1: MPASM ASSEMBLER WINDOWS SHELL INTERFACE}

Select a source file by typing in the name or using the \textbf{Browse} button. Set the various options as described below. (Default options are read from the source file.) Then click \textbf{Assemble} to assemble the source file.

\textbf{Note:} When MPASM assembler for Windows is invoked through MPLAB IDE, this options screen is not available. Use the MPASM Assembler tab of the Build Options dialog in MPLAB IDE (\textit{Project>Build Options>Project}) to set options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radix</td>
<td>Override any source file radix settings.</td>
</tr>
<tr>
<td>Warning Level</td>
<td>Override any source file message level settings.</td>
</tr>
<tr>
<td>Hex Output</td>
<td>Override any source file hex file format settings.</td>
</tr>
<tr>
<td>Generated Files</td>
<td>Enable/disable various output files.</td>
</tr>
<tr>
<td>Case Sensitivity</td>
<td>Enable/disable case sensitivity.</td>
</tr>
<tr>
<td>Tab Size</td>
<td>Set the list file tab size.</td>
</tr>
<tr>
<td>Macro Expansion</td>
<td>Override any source file macro expansion settings.</td>
</tr>
</tbody>
</table>

References:
- Section 4.43 “\textit{list} - Listing Options”
- Section 4.56 “\textit{radix} - Specify Default Radix”
- Section 3.4 “Numeric Constants and Radix”
- Section 4.48 “\textit{messg} - Create User Defined Message”
- Section 1.7.5 “Hex File Formats (.hex, .hxl, .hxh)”
- Section 1.7 “Assembler Input/Output Files”
- Section 1.7.3 “Listing File (.lst)”
- Section 4.32 “expand - Expand Macro Listing”
2.4 COMMAND LINE INTERFACE

MPASM assembler can be invoked through the command line interface (command prompt) as follows:

```
mpasmwin [/option1.../optionN] filename
```

where

`/option` - refers to one of the command line options

`filename` - is the file being assembled

For example, if `test.asm` exists in the current directory, it can be assembled with the following command:

```
mpasmwin /e /l test.asm
```

If the source filename is omitted, the appropriate shell interface is invoked, i.e., a Windows OS interface is displayed, which includes a Help button.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Override any source file processor settings.</td>
<td></td>
</tr>
<tr>
<td>Extended Mode</td>
<td>Enable PIC18F extended instruction support.</td>
<td></td>
</tr>
<tr>
<td>Extra Options</td>
<td>Any additional command-line options. Reference: Section 2.4 “Command Line Interface”</td>
<td></td>
</tr>
<tr>
<td>Save Settings on Exit</td>
<td>Save these settings in mplab.ini. They will be used the next time you run mpasmwin.exe.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/?</code></td>
<td>N/A</td>
<td>Display the assembler help screen.</td>
</tr>
<tr>
<td><code>/ahex-format</code></td>
<td>INHX32*</td>
<td>Generate .hex output directly from assembler, where <code>hex-format</code> is one of {INHX8M</td>
</tr>
<tr>
<td><code>/c</code></td>
<td>On</td>
<td>Enable/Disable case sensitivity. If enabled, the assembler will distinguish between upper- and lower-case letters.</td>
</tr>
<tr>
<td><code>/d</code></td>
<td>N/A</td>
<td>Define a text string substitution, i.e., assign <code>value</code> to <code>label</code>.</td>
</tr>
<tr>
<td>`/e[+</td>
<td>-]`</td>
<td>On</td>
</tr>
<tr>
<td><code>/h</code></td>
<td>N/A</td>
<td>Display the assembler help screen.</td>
</tr>
<tr>
<td>`/l[+</td>
<td>-]`</td>
<td>On</td>
</tr>
<tr>
<td>Option</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>/m[+</td>
<td>-]</td>
<td>On</td>
</tr>
</tbody>
</table>
/o Enable       
/o+ Enable     
/o- Disable     
/o path Enable/specify path See Section 1.7.7 “Object File (.o)” for more information. |
| /processor_type     | None    | Set the processor type, where processor_type is a PIC1X MCU device, e.g., PIC18F452. |
| /q[+| -]             | Off     | Enable/Disable quiet mode (suppress screen output.) |
| /rradix              | Hex     | Defines default radix, where radix is one of {HEX | DEC | OCT}. See Section 4.43 “list - Listing Options” or Section 4.56 “radix - Specify Default Radix” for more information. |
| /t                   | 8       | Set the size of tabs in the list file. See Section 1.7.3 “Listing File (.lst)” for more information. |
| /wvalue              | 0       | Set message level, where value is one of {0|1|2}. 
0 all messages 
1 errors and warnings 
2 errors only See Section 4.48 “messg - Create User Defined Message” for more information. |
/x Enable       
/x+ Enable     
/x- Disable     
/x path Enable/specify path See Section 1.7.6 “Cross Reference File (.xrf)” for more information. |
| /y[+| -]             | Disabled| Enable/Disable extended instruction set. 
/y Enable       
/y+ Enable     
/y- Disable |
Can only be enabled for processors which support the extended instruction set and for the generic processor PIC18CXXX. /y- overrides LIST PE=type directive (see Section 4.43 “list -Listing Options”.) |

* Default is dependent on processor selected.
Chapter 3. Expression Syntax and Operation

3.1 INTRODUCTION

Various expression formats, syntax, and operations used by MPASM assembler are described here.

Topics covered in this chapter:

• Text Strings
• Reserved Words and Section Names
• Numeric Constants and Radix
• Arithmetic Operators and Precedence

3.2 TEXT STRINGS

A “string” is a sequence of any valid ASCII character (of the decimal range of 0 to 127) enclosed by 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.

Strings may be of any length that will fit within a 255 column source line. If a matching quote mark is found, the string ends. If none is found before the end of the line, the string will end at the end of the line. While there is no direct provision for continuation onto a second line, it is generally no problem to use a second dw directive for the next line.

The dw directive will store the entire string into successive words. If a string has an odd number of characters (bytes), the dw and data directives will pad the end of the string with one byte of zero (00).

If a string is used as a literal operand, it must be exactly one character long, or an error will occur.
3.2.1 Escape Characters

The assembler accepts the ANSI ‘C’ escape sequences to represent certain special control characters:

<table>
<thead>
<tr>
<th>Escape Character</th>
<th>Description</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Bell (alert) character</td>
<td>07</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace character</td>
<td>08</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed character</td>
<td>0C</td>
</tr>
<tr>
<td>\n</td>
<td>New line character</td>
<td>0A</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return character</td>
<td>0D</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab character</td>
<td>09</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab character</td>
<td>0B</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash</td>
<td>5C</td>
</tr>
<tr>
<td>?</td>
<td>Question mark character</td>
<td>3F</td>
</tr>
<tr>
<td>'</td>
<td>Single quote (apostrophe)</td>
<td>27</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote character</td>
<td>22</td>
</tr>
<tr>
<td>\000</td>
<td>Octal number (zero, Octal digit, Octal digit)</td>
<td></td>
</tr>
<tr>
<td>\xHH</td>
<td>Hexadecimal number</td>
<td></td>
</tr>
</tbody>
</table>

3.2.2 Code Examples

See the examples below for the object code generated by different statements involving strings.

```
dw "testing output string one\n"
```

```
7465 7374 696E                    
6720 6F75 7470
7574 2073 7472
696E 6720 6F6E                   
650A

#define str "testing output string two"
```

```
B061                   
7465 7374 696E             
6720 6669 7273
7420 6F75 7470
7574 2073 7472
696E 6700
```
3.3 RESERVED WORDS AND SECTION NAMES

You may not use the following words for label, constant or variable names:

- Directives (see Chapter 4, “Directives”).
- Instructions (see Appendix A, “Instruction Sets”).
- The word “main” (when using the assembler with MPLAB IDE). Do not use a “main” label that cannot be reached by a simple reset and run, for example, a data label named “main” or a routine named “main” that will only be accessed under certain conditions, as with an interrupt.

In addition, the assembler has the following reserved section names:

<table>
<thead>
<tr>
<th>Section Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>.access_ovr</td>
<td>Default section name for access_ovr directive.</td>
</tr>
<tr>
<td>.code</td>
<td>Default section name for code directive.</td>
</tr>
<tr>
<td>.idata</td>
<td>Default section names for idata and idata_acs directives, respectively.</td>
</tr>
<tr>
<td>.idata_acs</td>
<td></td>
</tr>
<tr>
<td>.udata</td>
<td>Default section names for udata, udata_acs, udata_ovr and udata_shr directives, respectively.</td>
</tr>
<tr>
<td>.udata_acs</td>
<td></td>
</tr>
<tr>
<td>.udata_ovr</td>
<td></td>
</tr>
<tr>
<td>.udata_shr</td>
<td></td>
</tr>
</tbody>
</table>

3.4 NUMERIC CONSTANTS AND RADIX

MPASM assembler supports the following radix forms for constants: hexadecimal, decimal, octal, binary, and ASCII. The default radix is hexadecimal; the default radix determines what value will be assigned to constants in the object file when a radix is not explicitly specified by a base descriptor.

**Note:** The radix for numeric constants can be made different from the default radix specified with the directives `radix` or `list r=`. Also, allowable default radices are limited to hexadecimal, decimal, and octal.

Constants can be optionally preceded by a plus or minus sign. If unsigned, the value is assumed to be positive.

**Note:** Intermediate values in constant expressions are treated as 32-bit unsigned integers. Whenever an attempt is made to place a constant in a field for which it is too large, a truncation warning will be issued.
The following table presents the various radix specifications:

**TABLE 4: RADIX SPECIFICATIONS - MPASM ASSEMBLER/MPLINK LINKER**

<table>
<thead>
<tr>
<th>Note</th>
<th>Type</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Binary</td>
<td>B'binary_digits'</td>
<td>B'00111001'</td>
</tr>
<tr>
<td>2</td>
<td>Octal</td>
<td>O'octal_digits'</td>
<td>O'777'</td>
</tr>
<tr>
<td>3</td>
<td>Decimal</td>
<td>D'digits' .digits</td>
<td>D'100' .100</td>
</tr>
<tr>
<td>4</td>
<td>Hexadecimal</td>
<td>H'hex_digits' 0xhex_digits</td>
<td>H'9f' 0x9f</td>
</tr>
<tr>
<td>5</td>
<td>ASCII</td>
<td>A'character' 'character'</td>
<td>A'C' 'C'</td>
</tr>
</tbody>
</table>

1. A binary integer is 'b' or 'B' followed by one or more of the binary digits '01' in single quotes.
2. An octal integer is 'o' or 'O' followed by one or more of the octal digits '01234567' in single quotes.
3. A decimal integer is 'd' or 'D' followed by one or more decimal digits '0123456789' in single quotes. Or, a decimal integer is '.' followed by one or more decimal digits '0123456789'.
4. A hexadecimal integer is 'h' or 'H' followed by one or more hexadecimal digits '0123456789abcdefABCDEF' in single quotes. Or, a hexadecimal integer is '0x' or '0X' followed by one or more hexadecimal digits '0123456789abcdefABCDEF'.
5. An ASCII character is 'a' or 'A' followed by one character (see Section B.2 "ASCII Character Set") in single quotes. Or, an ASCII character is one character in single quotes.

3.5 ARITHMETIC OPERATORS AND PRECEDENCE

Arithmetic operators may be used with directives and their variables as specified in the table below.

**Note:** These operators cannot be used with program variables. They are for use with directives only.

The operator order in the table also corresponds to its precedence, where the first operator has the highest precedence and the last operator has the lowest precedence. Precedence refers to the order in which operators are executed in a code statement.

**TABLE 3-1: ARITHMETIC OPERATORS IN ORDER OF PRECEDENCE**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>goto $ + 3</td>
</tr>
<tr>
<td>(</td>
<td>1 + (d * 4)</td>
</tr>
<tr>
<td>)</td>
<td>(Length + 1) * 256</td>
</tr>
<tr>
<td>!</td>
<td>if ! (a == b)</td>
</tr>
<tr>
<td>-</td>
<td>-1 * Length</td>
</tr>
<tr>
<td>~</td>
<td>flags = ~flags</td>
</tr>
<tr>
<td>low¹</td>
<td>movlw low CTR_Table</td>
</tr>
<tr>
<td>high¹</td>
<td>movlw high CTR_Table</td>
</tr>
<tr>
<td>upper¹</td>
<td>movlw upper CTR_Table</td>
</tr>
<tr>
<td>*</td>
<td>a = b * c</td>
</tr>
<tr>
<td>/</td>
<td>a = b / c</td>
</tr>
<tr>
<td>%</td>
<td>entry_len = tot_len % 16</td>
</tr>
<tr>
<td>+</td>
<td>tot_len = entry_len * 8 + 1</td>
</tr>
<tr>
<td>-</td>
<td>entry_len = (tot - 1) / 8</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>flags = flags &lt;&lt; 1</td>
</tr>
</tbody>
</table>
### TABLE 3-1: ARITHMETIC OPERATORS IN ORDER OF PRECEDENCE

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;</td>
<td><code>flags = flags &gt;&gt; 1</code></td>
</tr>
<tr>
<td>&gt;=</td>
<td><code>if entry_idx &gt;= num_entries</code></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>if entry_idx &gt; num_entries</code></td>
</tr>
<tr>
<td>&lt;</td>
<td><code>if entry_idx &lt; num_entries</code></td>
</tr>
<tr>
<td>&lt;=</td>
<td><code>if entry_idx &lt;= num_entries</code></td>
</tr>
<tr>
<td>==</td>
<td><code>if entry_idx == num_entries</code></td>
</tr>
<tr>
<td>!=</td>
<td><code>if entry_idx != num_entries</code></td>
</tr>
<tr>
<td>&amp;</td>
<td><code>flags = flags &amp; ERROR_BIT</code></td>
</tr>
<tr>
<td>^</td>
<td><code>flags = flags ^ ERROR_BIT</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td><code>if (len == 512) &amp;&amp; (b == c)</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td><code>entry_index = 0</code></td>
</tr>
<tr>
<td>+=</td>
<td><code>entry_index += 1</code></td>
</tr>
<tr>
<td>-=</td>
<td><code>entry_index -= 1</code></td>
</tr>
<tr>
<td>*=</td>
<td><code>entry_index *= entry_length</code></td>
</tr>
<tr>
<td>/=</td>
<td><code>entry_total /= entry_length</code></td>
</tr>
<tr>
<td>%=</td>
<td><code>entry_index %= 8</code></td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td><code>flags &lt;&lt;= 3</code></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td><code>flags &gt;&gt;= 3</code></td>
</tr>
<tr>
<td>&amp;=</td>
<td><code>flags &amp;= ERROR_FLAG</code></td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>^=</td>
<td><code>flags ^= ERROR_FLAG</code></td>
</tr>
<tr>
<td>++</td>
<td><code>i ++</code></td>
</tr>
<tr>
<td>--</td>
<td><code>i --</code></td>
</tr>
</tbody>
</table>

**Note 1:** This precedence is the same for the low, high and upper operands which apply to sections. See Section 6.4 “Low, High and Upper Operators” for more information.

**Note 2:** These operators can only be used on a line by themselves; they cannot be embedded within other expression evaluations.
Chapter 4. Directives

4.1 INTRODUCTION

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

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

Many of the assembler directives have alternate names and formats. These may exist to provide backward compatibility with previous assemblers from Microchip and to be compatible with individual programming practices. If portable code is desired, it is recommended that programs be written using the specifications contained here.

**Note:** Although MPASM assembler is often used with MPLINK object linker, MPASM assembler directives are not supported in MPLINK linker scripts. See MPLINK object linker documentation for more information on linker options to control listing and hex file output.

Information on individual directives includes syntax, description, usage, and related directives, as well as simple and, in some cases, expanded examples of use. In most cases, simple examples may be assembled and run by adding an `end` statement. Expanded examples may be assembled and run as-is to give an demonstration of an application using the directive(s).

Individual directives may be found alphabetically (in the following sections) or by type (Section 4.2 “Directives by Type”).

**Note:** Directives are not case-sensitive, e.g., cblock may be executed as CBLOCK, cblock, Cblock, etc

4.2 DIRECTIVES BY TYPE

There are six basic types of directives provided by the assembler.

1. Control Directives
2. Conditional Assembly Directives
3. Data Directives
4. Listing Directives
5. Macro Directives
6. Object File Directives
4.2.1 Control Directives

Control directives control how code is assembled.

- `#define` - Define a Text Substitution Label .............................................. p. 65
- `#include` - Include Additional Source File ............................................... p. 90
- `#undefine` - Delete a Substitution Label.................................................. p. 116
- `bankisel` - Generate Indirect Bank Selecting Code (PIC12/16 MCUs) ... p. 48
- `banksel` - Generate Bank Selecting Code ............................................... p. 50
- `constant` - Declare Symbol Constant ...................................................... p. 58
- `end` - End Program Block........................................................................... p. 69
- `equ` - Define an Assembler Constant......................................................... p. 71
- `org` - Set Program Origin........................................................................... p. 99
- `pagesel` - Generate Page Selecting Code (PIC10/12/16 MCUs)............. p. 102
- `pageselw` - Generate Page Selecting Code Using WREG Commands (PIC10/12/16 MCUs) .............................................................. p. 103
- `processor` - Set Processor Type ............................................................. p. 104
- `radix` - Specify Default Radix ................................................................... p. 105
- `set` - Define an Assembler Variable .......................................................... p. 108
- `variable` - Declare Symbol Variable ....................................................... p. 117

4.2.2 Conditional Assembly Directives

Conditional assembly directives permit sections of conditionally assembled code. These are not run-time instructions like their C language counterparts. They define which code is assembled, not how the code executes.

- `else` - Begin Alternative Assembly Block to if Conditional ..................... p. 68
- `endif` - End Conditional Assembly Block ................................................. p. 70
- `endw` - End a while Loop .......................................................................... p. 71
- `if` - Begin Conditionally Assembled Code Block....................................... p. 86
- `ifdef` - Execute If Symbol has Been Defined........................................... p. 88
- `ifndef` - Execute If Symbol has not Been Defined .................................. p. 89
- `while` - Perform Loop While Condition is True ........................................ p. 118

4.2.3 Data Directives

Data directives control the allocation of memory and provide a way to refer to data items symbolically, i.e., by meaningful names.

- `__badram` - Identify Unimplemented RAM ................................................ p. 46
- `__badrom` - Identify Unimplemented ROM ................................................ p. 47
- `__config` - Set Processor Configuration Bits............................................ p. 55
- `config` - Set Processor Configuration Bits (PIC18 MCUs) ....................... p. 57
- `__idlocs` - Set Processor ID Locations .................................................... p. 85
- `__maxram` - Define Maximum RAM Location ........................................... p. 95
- `__maxrom` - Define Maximum ROM Location .......................................... p. 96
- `cblock` - Define a Block of Constants....................................................... p. 52
- `da` - Store Strings in Program Memory (PIC12/16 MCUs) ....................... p. 59
- `data` - Create Numeric and Text Data ....................................................... p. 60
- `db` - Declare Data of One Byte ................................................................ p. 62
- `de` - Declare EEPROM Data Byte .......................................................... p. 64
- `dt` - Define Table (PIC12/16 MCUs)......................................................... p. 67
Directives

4.2.4 Listing Directives
Listing directives control the MPASM assembler listing file format. These directives allow the specification of titles, pagination, and other listing control. Some listing directives also control how code is assembled.

- **dtm** - Define Table (Extended PIC16 MCUs Only) ................................................p. 67
- **dw** - Declare Data of One Word ...........................................................................p. 68
- **endc** - End an Automatic Constant Block .........................................................p. 69
- **fill** - Specify Program Memory Fill Value ..............................................................p. 80
- **res** - Reserve Memory .........................................................................................p. 106

4.2.5 Macro Directives
Macro directives control the execution and data allocation within macro body definitions.

- **endm** - End a Macro Definition ........................................................................p. 70
- **exitm** - Exit from a Macro ................................................................................p. 75
- **expand** - Expand Macro Listing ........................................................................p. 77
- **local** - Declare Local Macro Variable ...............................................................p. 92
- **macro** - Declare Macro Definition .....................................................................p. 94
- **noexpand** - Turn off Macro Expansion ...............................................................p. 98

4.2.6 Object File Directives
Object file directives are used only when creating an object file.

- **access_ovr** - Begin an Object File Overlay Section in Access RAM (PIC18 MCUs) ................................................................................................................p. 46
- **code** - Begin an Object File Code Section .................................................................p. 54
- **code_pack** - Begin an Object File Packed Code Section (PIC18 MCUs). p. 55
- **extern** - Declare an Externally Defined Label .......................................................p. 78
- **idata** - Begin an Object File Initialized Data Section ..............................................p. 82
- **idata_acs** - Begin an Object File Initialized Data Section in Access RAM (PIC18 MCUs) ..............................................................................................................p. 84
- **udata** - Begin an Object File Uninitialized Data Section ........................................p. 110
- **udata_acs** - Begin an Object File Access Uninitialized Data Section (PIC18 MCUs) ......................................................................................................................p. 111
- **udata_ovr** - Begin an Object File Overlaid Uninitialized Data Section ....p. 113
- **udata_shr** - Begin an Object File Shared Uninitialized Data Section (PIC12/16 MCUs) ......................................................................................................................p. 115
4.3 access_ovr - BEGIN AN OBJECT FILE OVERLAY SECTION IN ACCESS RAM (PIC18 MCUs)

4.3.1 Syntax

[label] access_ovr [RAM_address]

4.3.2 Description

This directive declares the beginning of a section of overlay data in Access RAM. If label is not specified, the section is named .access_ovr. The starting address is initialized to the specified address or will be assigned at link time if no address is specified. The space declared by this section is overlaid by all other access_ovr sections of the same name. No code can be placed by the user in this segment.

4.3.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”. access_ovr is similar to udata_acs and udata_ovr, except that it declares a PIC18 Access-RAM, uninitialized-data section that can be overlaid with other overlay access sections of the same name. Overlaying access sections allows you to reuse access-bank data space.

4.3.4 See Also

extern global udata udata_ovr udata_acs

4.3.5 Simple Example

;The 2 identically-named sections are overlaid in PIC18 Access RAM.
;In this example, u16a is overlaid with memory locations used
;by u8a and u8b. u16b is overlaid with memory locations used
;by u8c and u8d.

myaoscn       access_ovr
u8a:          res 1
u8b:          res 1
u8c:          res 1
u8d:          res 1

myaoscn       access_ovr
u16a:         res 2
u16b:         res 2

4.4 __badram - IDENTIFY UNIMPLEMENTED RAM

Note: badram is preceded by two underline characters.

4.4.1 Syntax

__badram expr[-expr], expr[-expr]
4.4.2 Description

The __maxram and __badram directives together flag accesses to unimplemented registers. __badram defines the locations of invalid RAM addresses. This directive is designed for use with the __maxram directive. A __maxram directive must precede any __badram directive. Each expr must be less than or equal to the value specified by __maxram. Once the __maxram directive is used, strict RAM address checking is enabled, using the RAM map specified by __badram. To specify a range of invalid locations, use the syntax minloc - maxloc.

4.4.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”. __badram is not commonly used, as RAM and ROM details are handled by the include files (*.inc) or linker script files (*.lkr).

4.4.4 See Also

__maxram

4.4.5 Simple Example

#include p16c622.inc
__maxram 0x0BF
__badram 0x07-0x09, 0x0D-0xE
__badram 0x87-0x89, 0x8D, 0x8F-0x9E
movwf 0x07 ; Generates invalid RAM warning
movwf 0x87 ; Generates invalid RAM warning ; and truncation message

4.5 __badrom - IDENTIFY UNIMPLEMENTED ROM

Note: badrom is preceded by two underline characters.

4.5.1 Syntax

__badrom expr[-expr] [, expr[-expr]]

4.5.2 Description

The __maxrom and __badrom directives together flag accesses to unimplemented registers. __badrom defines the locations of invalid ROM addresses. This directive is designed for use with the __maxrom directive. A __maxrom directive must precede any __badrom directive. Each expr must be less than or equal to the value specified by __maxrom. Once the __maxrom directive is used, strict ROM address checking is enabled, using the ROM map specified by __badrom. To specify a range of invalid locations, use the syntax minloc - maxloc.

Specifically, a warning will be raised in the following circumstances:

• the target of a GOTO or CALL instruction is evaluated by the assembler to a constant, and falls in a bad ROM region
• the target of an LGOTO or LCALL pseudo-op is evaluated by the assembler to a constant, and falls in a bad ROM region
• a .hex file is being generated, and part of an instruction falls in a bad ROM region
4.5.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

__badrom is not commonly used, as RAM and ROM details are handled by the include files (*.inc) or linker script files (*.lkr).

4.5.4 See Also

__maxrom

4.5.5 Simple Example

#include p12c508.inc
__maxrom 0x1FF
__badrom 0x2 - 0x4, 0xA
org 0x5
  goto 0x2 ; generates a warning
call 0x3 ; generates a warning
org 0xA
  movlw 5 ; generates a warning

4.6 bankisel - GENERATE INDIRECT BANK SELECTING CODE (PIC12/16 MCUs)

4.6.1 Syntax

bankisel label

4.6.2 Description

This directive is an instruction to the assembler or linker to generate the appropriate bank selecting code for an indirect access of the register address specified by label. Only one label should be specified. No operations can be performed on label. This label must have been previously defined.

The linker will generate the appropriate bank selecting code. For 14-bit instruction width (most PIC12/PIC16) devices, the appropriate bit set/clear instruction on the IRP bit in the STATUS register will be generated. But for PIC16 extended instructions, FSR0H is modified instead of IRP bit (as there is no IRP bit). If the indirect address can be specified without these instructions, no code will be generated.

4.6.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive may be used with 14-bit instruction width PIC1X devices. This excludes 12-bit instruction width devices and PIC18 devices.

4.6.4 See Also

bankisel pagesel

4.6.5 Simple Example

movlw Var1
movwf FSR ;Load the address of Var1 info FSR
bankisel Var1 ;Select the correct bank for Var1
  :movwf INDF ;Indirectly write to Var1
### 4.6.6 Application Example - bankisel

This program demonstrates the `bankisel` directive. This directive generates the appropriate code to set/clear the IRP bit of the STATUS register for an indirect access.

```c
#include p16f877a.inc ;Include standard header file ;for the selected device.

group1 udata 0x20 ;group1 data stored at locations ;starting at 0x20 (IRP bit 0).
   group1_var1 res 1 ;group1_var1 located at 0x20.
   group1_var2 res 1 ;group1_var2 located at 0x21.

group2 udata 0x120 ;group2 data stored at locations ;starting at 0x120 (IRP bit 1).
   group2_var1 res 1 ;group2_var1 located at 0x120.
   group2_var2 res 1 ;group2_var2 located at 0x121.

RST CODE 0x0 ;The code section named RST ;is placed at program memory ;location 0x0. The next two ;instructions are placed in ;code section RST.
   pagesel start ;Jumps to the location labelled ;'start'.
   goto start ;'start'.

PGM CODE ;This is the beginning of the ;code section named PGM. It is ;a relocatable code section ;since no absolute address is ;given along with directive CODE.

start
   movlw 0x20 ;This part of the code addresses
   movwf FSR ;variables group1_var1 &
   bankisel group1_var1 ;group1_var2 indirectly.
   clrf INDF
   incf FSR,F
   clrf INDF

   movwf FSR
   bankisel group2_var1
   clrf INDF
   incf FSR,F
   clrf INDF

   goto $ ;Go to current line (loop here)
end
```
4.6.7 Application Example 2 - bankisel

```asm
#include pl6f877a.inc ;Include standard header file
;for the selected device.

bankisel EEADR ;This register is at location 100h
;in banks 2 or 3 so the IRP bit
;must be set. bankisel will set it
;but only where it is used.

movlw EEADR,W ;Put the address of the register to
;be accessed indirectly into W.

movwf FSR ;Copy address from W to FSR to set
;up pointer to EEADR.

crfl INDW ;Clear EEADR through indirect
;accessing of EEADR through FSR/INDF.
;It would have cleared PIR2 (00Dh)
;if bankisel had not been used to
;set the IRP bit.

goto $ ;Prevents fall off end of code.
end ;All code must have an end statement.
```

4.7 banksel - GENERATE BANK SELECTING CODE

4.7.1 Syntax

`banksel label`

4.7.2 Description

This directive is an instruction to the assembler and linker to generate bank selecting code to set the bank to the bank containing the designated `label`. Only one `label` should be specified. No operations can be performed on `label`. This label must have been previously defined.

The linker will generate the appropriate bank selecting code:

- For 12-bit instruction width (PIC10F, some PIC12/PIC16) devices, the appropriate bit set/clear instructions on the FSR will be generated.
- For 14-bit instruction width (most PIC12/PIC16) devices, bit set/clear instructions on the STATUS register will be generated.
- For PIC16 extended and PIC18 devices, a `movlb` will be generated. If the device contains only one bank of RAM, no instructions will be generated.

4.7.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive may be used with all PIC1X devices. This directive is not needed for variables in access RAM (PIC18 devices.)

4.7.4 See Also

`bankisel pagesel`

4.7.5 Simple Example

```asm
banksel Var1 ;Select the correct bank for Var1
movwf Var1 ;Write to Var1
```
4.7.6 Application Example - banksel

This program demonstrates the banksel directive. This directive generates the appropriate code to set/clear the RP0 and RP1 bits of the STATUS register.

```
#include p16f877a.inc    ;Include standard header file
                   ;for the selected device.

group1  udata 0x20    ;group1 data stored at locations
                      ;starting at 0x20 (bank 0).
group1_var1 res 1    ;group1_var1 located at 0x20.
group1_var2 res 1    ;group1_var2 located at 0x21.

group2  udata 0xA0    ;group2 data stored at locations
                      ;starting at 0xA0 (bank 1)
group2_var1 res 1
group2_var2 res 1

RST     CODE     0x0    ;The code section named RST
                          ;is placed at program memory
                          ;location 0x0. The next two
                          ;instructions are placed in
                          ;code section RST.

pagesel  start      ;Jumps to the location labelled
goto     start      ;'start'.

PGM     CODE             ;This is the beginning of the
                          ;code section named PGM. It is
                          ;a relocatable code section
                          ;since no absolute address is
                          ;given along with directive CODE.

start
banksel  group1_var1   ;This directive generates code
                       ;to set/clear bank select bits
                       ;RP0 & RP1 of STATUS register
                       ;depending upon the address of
                       ;group1_var1.
clrf  group1_var1
clrf  group1_var2

banksel  group2_var1   ;This directive generates code
                       ;to set/clear bank select bits
                       ;RP0 & RP1 of STATUS register
                       ;depending upon the address of
                       ;group2_var1.
clrf  group2_var1
clrf  group2_var2

goto  $                ;Go to current line (loop here)
end
```
4.7.7  Application Example 2 - \texttt{banksel}

\begin{verbatim}
#include pl6f877a.inc ;Include standard header file ;for the selected device.

banksel TRISB ;Since this register is in bank 1, 
    ;not default bank 0, banksel is 
    ;used to ensure bank bits are correct.

clrf TRISB ;Clear TRISB. Sets PORTB to outputs.

banksel PORTB ;banksel used to return to bank 0, 
    ;where PORTB is located.

movlw 0x55 ;Set PORTB value.

movwf PORTB

goto $ ;All programs must have an end.
\end{verbatim}

4.8  \texttt{cblock} - \textbf{DEFINE A BLOCK OF CONSTANTS}

4.8.1  Syntax

\begin{verbatim}
cblock \[expr\] label[:increment] [, label[:increment]]
endc
\end{verbatim}

4.8.2  Description

Defines a list of named sequential symbols. The purpose of this directive is to assign address offsets to many labels. The list of names end when an \texttt{endc} directive is encountered.

\texttt{expr} indicates the starting value for the first name in the block. If no expression is found, the first name will receive a value one higher than the final name in the previous \texttt{cblock}. If the first \texttt{cblock} in the source file has no \texttt{expr}, assigned values start with zero.

If \texttt{increment} is specified, then the next \texttt{label} is assigned the value of \texttt{increment} higher than the previous \texttt{label}.

Multiple names may be given on a line, separated by commas.

\texttt{cblock} is useful for defining constants in program and data memory for absolute code generation.

4.8.3  Usage

This directive is used in the following types of code: absolute. For information on types of code, see Section 1.6 \textbf{“Assembler Operation”}.

Use this directive in place of or in addition to the \texttt{equ} directive. When creating non-relocatable (absolute) code, \texttt{cblock} is often used to define variable address location names. Do not use \texttt{cblock} or \texttt{equ} to define variable location names for relocatable code.

4.8.4  See Also

\begin{verbatim}
endc equ
\end{verbatim}
4.8.5 Simple Example

cblock 0x20 ; name_1 will be assigned 20
    name_1, name_2 ; name_2, 21 and so on
    name_3, name_4 ; name_4 is assigned 23.
endc

cblock 0x30
    TwoByteVar: 0, TwoByteHigh, TwoByteLow ; TwoByteVar = 0x30
    ; TwoByteHigh = 0x30
    ; TwoByteLow = 0x31
endc

4.8.6 Application Example - cblock/endc

This example shows the usage of CBLOCK and ENDC directives for defining constants or variables in data memory space. The same directives can be used for program memory space also.

The program calculates the perimeter of a rectangle. Length and width of the rectangle will be stored in buffers addressed by length (22H) and width (23H). The calculated perimeter will be stored in the double-precision buffer addressed by perimeter (i.e., 20H and 21H).

#include p16f877a.inc ; Include standard header file ; for the selected device.

CBLOCK 0x20 ; Define a block of variables ; starting at 20H in data memory.
    perimeter:2 ; The label perimeter is 2 bytes ; wide. Address 20H and 21H is ; assigned to the label perimeter.
    length ; Address 22H is assigned to the ; label length.
    width ; Address 23H is assigned to the ; label width.
ENDC ; This directive must be supplied ; to terminate the CBLOCK list.

cclf perimeter+1 ; Clear perimeter high byte ; at address 21H.
movf length,w ; Move the data present in the ; register addressed by 'length' ; to 'w'.
addwf width,w ; Add data in 'w' with data in the ; register addressed by 'width'. ; STATUS register carry bit C ; may be affected.
movwf perimeter ; Move 'w' to the perimeter low ; byte at address 20H. Carry bit ; is unaffected.
rlf perimeter+1 ; Increment register 21H if carry ; was generated. Also clear carry ; if bit was set.
rlf perimeter ; Multiply register 20H by 2. ; Carry bit may be affected.
rlf perimeter+1 ; Again, increment register 21H ; if carry was generated.
goto $ ; Go to current line (loop here)
end
4.9 code - BEGIN AN OBJECT FILE CODE SECTION

4.9.1 Syntax
[label] code [ROM_address]

4.9.2 Description
This directive declares the beginning of a section of program code. If label is not specified, the section is named .code. The starting address is initialized to the specified address or will be assigned at link time if no address is specified.

Note: Two sections in a source file may not have the same name.

4.9.3 Usage
This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

There is no “end code” directive. The code of a section ends automatically when another code or data section is defined or when the end of the file is reached.

4.9.4 See Also
extern code_pack global idata udata udata_acs udata_cvr udata_shr

4.9.5 Simple Example
RESET code 0x01FF
goto START

4.9.6 Application Example - code
This program demonstrates the code directive, which declares the beginning of a section of program code.

#include p16f877a.inc ;Include standard header file
;for the selected device.

RST CODE 0x0 ;The code section named RST
;is placed at program memory
;location 0x0. The next two
;instructions are placed in
;code section RST.

pagesel start goto start ;Jumps to the location labelled
;'start'.

PGM CODE ;This is the beginning of the
;code section named PGM. It is
;a relocatable code section
;since no absolute address is
;given along with directive CODE.

start
clrw
goto $ ;Go to current line (loop here)

nop ;This is a relocatable code
;section since no address is
;specified. The section name will
;be, by default, .code.

dw
4.10  code_pack - BEGIN AN OBJECT FILE PACKED CODE SECTION (PIC18 MCUs)

4.10.1 Syntax

[label] code_pack [ROM_address]

4.10.2 Description

This directive declares the beginning of a section of program code or ROM data where a padding byte of zero is not appended to an odd number of bytes. If label is not specified, the section is named .code. The starting address is initialized to ROM_address or will be assigned at link time if no address is specified. If ROM_address is specified, it must be word-aligned. If padded data is desired, use db.

Note: Two sections in a source file may not have the same name

4.10.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is commonly used when storing data into program memory (use with db) or the EEPROM data memory (use with de) of a PIC18 device.

4.10.4 See Also

extern code global idata udata udata_acs udata_ovr udata_shr

4.10.5 Simple Example

00001 LIST P=18Cxx
00002
00003 packed code_pack 0x1F0
0001F0 01 02 03       00004   DB 1, 2, 3
0001F3 04 05          00005   DB 4, 5
00006
00007 padded code
000000 0201 0003      00008   DB 1, 2, 3
000004 0504           00009   DB 4, 5
00010
00011   END

4.11  __config - SET PROCESSOR CONFIGURATION BITS

Note: config is preceded by two underline characters.

4.11.1 Syntax

Preferred:
__config expr
__config addr, expr

Note: PIC18FXXJ devices do not support this directive. Use config directive (no underline characters.)

Supported:
__fuses expr
4.11.2 Description

Sets the processor's configuration bits. Before this directive is used, the processor must be declared through the command line, the list directive, the processor directive or Configure>Select Device if using MPLAB IDE. Refer to individual PIC1X microcontroller data sheets for a description of the configuration bits.

**MCUs with a single configuration register**

Sets the processor's configuration bits to the value described by `expr`.

**MCUs with multiple configuration registers**

For the address of a valid configuration byte specified by `addr`, sets the configuration bits to the value described by `expr`.

**Note:** Configuration bits must be listed in ascending order.

Although this directive may be used to set configuration bits for PIC18 MCU devices, it is recommended that you use the `config` directive (no underline characters.) For PIC18FXXJ devices, you *must* use the `config` directive.

**Note:** Do not mix `__config` and `config` directives in the same code.

4.11.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is placed in source code so that, when the code is assembled into a hex file, the configuration values are preset to desired values in your application. This is useful when giving your files to a third-party programming house, as this helps insure the device is configured correctly when programmed.

Place configuration bit assignments at the beginning of your code. Use the configuration options (names) in the standard include (*.inc) file. These names can be bitwise ANDed together using & to declare multiple configuration bits.

4.11.4 See Also

`config __idlocs list processor`

4.11.5 Simple Examples

**Example 1: PIC16 Devices**

```plaintext
#include p16f877a.inc ;include file with config bit definitions
__config _HS_OSC & _WDT_OFF & _LVP_OFF ;Set oscillator to HS,
 ;watchdog time off,
 ;low-voltage prog. off
```

**Example 2: PIC17X Devices**

```plaintext
#include p17c42.inc ;include file with config bit definitions
__config 0xFFFF ;default configuration bits
```
Example 3: PIC18 Devices

```
#include p18c452.inc  ;Include standard header file
                ;for the selected device.

;code protect disabled.
__CONFIG _CONFIG0, _CP_OFF_0

;Oscillator switch disabled, RC oscillator with OSC2
;as I/O pin.
__CONFIG _CONFIG1, _OSCS_OFF_1 & _RCIO_OSC_1

;Brown-OutReset enabled, BOR Voltage is 2.5v
__CONFIG _CONFIG2, _BOR_ON_2 & _BORV_25_2

;Watch Dog Timer enable, Watch Dog Timer PostScaler
;count - 1:128
__CONFIG _CONFIG3, _WDT_ON_3 & _WDTPS_128_3

;CCP2 pin Mux enabled
__CONFIG _CONFIG5, _CCP2MX_ON_5

;Stack over/underflow Reset enabled
__CONFIG _CONFIG6, _STVR_ON_6
```

4.12 config - SET PROCESSOR CONFIGURATION BITS (PIC18 MCUs)

4.12.1 Syntax

```
cfg = setting=value [, setting=value]
```

4.12.2 Description

Defines a list of configuration bit setting definitions. This list sets the PIC18 processor's configuration bits represented by `setting` to a value described by `value`. Refer to individual PIC18 microcontroller data sheets for a description of the configuration bits. Available settings and values maybe found in both the standard processor include (*.inc) files and the PIC18 Configuration Settings Addendum (DS51537).

Multiple settings may be defined on a single line, separated by commas. Settings for a single configuration byte may also be defined on separate lines.

Before this directive is used, a PIC18 MCU must be declared through the command line, the list directive, the processor directive or Configure>Select Device in MPLAB IDE.

Another directive that may be used to set configuration bits for PIC18 MCU devices is the `__config` directive, but this is not recommended for new code.

**Note:** Do not mix `__config` and `config` directives in the same code.

4.12.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is placed in source code so that, when the code is compiled/assembled into a hex file, the configuration values are preset to desired values in your application. This is useful when giving your files to a third-party programming house, as this helps insure the device is configured correctly when programmed.
Place configuration bit assignments at the beginning of your code. Use the configuration options (\texttt{setting=value}) pairs listed in the standard include (*\texttt{.inc}) file or the addendum. The \texttt{config} directive can be used multiple times in the source code, but an error will be generated if the same bit is assigned a value more than once, i.e.,

\begin{verbatim}
CONFIG CP0=OFF, WDT=ON
CONFIG CP0=ON ;(An error will be issued since CP0 is assigned twice)
\end{verbatim}

\subsection*{4.12.4 \textbf{See Also}}

\texttt{\_config \_idlocs list processor}

\subsection*{4.12.5 \textbf{Simple Example}}

\begin{verbatim}
#include p18f452.inc ;Include standard header file for the selected device.

;code protect disabled
CONFIG CP0=OFF

;Oscillator switch enabled, RC oscillator with OSC2 as I/O pin.
CONFIG OSCS=ON, OSC=LP

;Brown-OutReset enabled, BOR Voltage is 2.5v
CONFIG BOR=ON, BORV=25

;Watch Dog Timer enable, Watch Dog Timer PostScaler count - 1:128
CONFIG WDT=ON, WDTPS=128

;CCP2 pin Mux enabled
CONFIG CCP2MUX=ON

;Stack over/underflow Reset enabled
CONFIG STVR=ON
\end{verbatim}

\section*{4.13 \texttt{constant} - \textbf{DECLARE SYMBOL CONSTANT}}

\subsection*{4.13.1 \textbf{Syntax}}

\texttt{constant \_label=expr [\ldots, \_label=\texttt{expr}]}

\subsection*{4.13.2 \textbf{Description}}

Creates symbols for use in MPASM assembler expressions. Constants may not be reset after having once been initialized, and the expression must be fully resolvable at the time of the assignment. This is the principal difference between symbols declared as constant and those declared as variable, or created by the \texttt{set} directive. Otherwise, constants and variables may be used interchangeably in absolute code expressions.

\subsection*{4.13.3 \textbf{Usage}}

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see \textit{Section 1.6 \textbf{Assembler Operation}}.

Although \texttt{equ} or \texttt{cblock} is more generally used to create constants, the \texttt{constant} directive also works.

\subsection*{4.13.4 \textbf{See Also}}

\texttt{set variable equ cblock}
4.13.5 Examples

See the examples under variable.

4.14 da - STORE STRINGS IN PROGRAM MEMORY (PIC12/16 MCUs)

4.14.1 Syntax

[label] da expr [, expr2, ..., exprn]

4.14.2 Description

da - Data ASCII.
Generates a packed 14-bit number representing two 7-bit ASCII characters.

4.14.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.
This directive is useful for storing strings in memory for PIC16 MCU devices.

4.14.4 Simple Examples

- da "abcdef"
  will put 30E2 31E4 32E6 into program memory
- da "12345678",0
  will put 18B2 19B4 1AB6 1BB8 0000 into program memory
- da 0xFFFF
  will put 0x3FFF into program memory

4.14.5 Application Example - da

This example shows the usefulness of directive da in storing a character string in the program memory of 14-bit architecture devices. This directive generates a packed 14-bit number representing two 7-bit ASCII characters.

```
#include p16f877a.inc ;Include standard header file
;for the selected device.

ORG 0x0000 ;The following code will be
           ;programmed in reset address 0.
goto start ;Jump to an address labelled
            ;'start'.

start ;Write your main program here.

goto $ ;Go to current line (loop here)

ORG 0x1000 ;Store the string starting from
           ;1000H.

Ch_stng da "PICmicro"
```

Directive `da` produces four 14-bit numbers: 2849, 21ED, 34E3, and 396F representing the ASCII equivalent of PI, Cm, ic, and ro. See below for more information.

```assembly
Sngl_ch da "A" ; 7-bit ASCII equivalents of 'A'
               ; and a NULL character will be packed
               ; in a 14-bit number.

da 0xff55 ; Places 3f55 in program memory.
               ; No packing.
end
```

### Determining 14-Bit Numbers

For the following statement:

```assembly
Ch_stng da "PICmicro"
```

The `da` directive produces four 14-bit numbers: 2849, 21ED, 34E3, and 396F representing the ASCII equivalent of PI, Cm, ic, and ro.

To see how the 14-bit numbers are determined, look at the ASCII values of P and I, which are 50h(01010000) and 49h(01001001) respectively. Each is presented in 7-bit as (0)1010000 and (0)1001001 respectively. The packed 14-bit number is 101000 01001001, which is stored as (00)101000 01001001 or 2849.

## 4.15 `data` - CREATE NUMERIC AND TEXT DATA

### 4.15.1 Syntax

```
[label] data expr[,expr,...,expr]
[label] data "text_string"[,"text_string",...]
```

### 4.15.2 Description

Initialize one or more words of program memory with data. The data may be in the form of constants, relocatable or external labels, or expressions of any of the above. The data may also consist of ASCII character strings, `text_string`, enclosed in single quotes for one character or double quotes for strings. Single character items are placed into the low byte of the word, while strings are packed two to a word. If an odd number of characters are given in a string, the final byte is zero. On all families except the PIC18 device family, the first character is in the most significant byte of the word. On the PIC18 device family, the first character is in the least significant byte of the word.

### 4.15.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

When generating a linkable object file, this directive can also be used to declare initialized data values. Refer to the `idata` directive for more information.

`db` and other data directives are more commonly used than `data`.

### 4.15.4 See Also

`db`, `de`, `dt`, `dtm`, `dw`, `idata`

### 4.15.5 Simple Example

```
data reloc_label+10 ; constants
data 1,2,ext_label ; constants, externals
data "testing 1,2,3" ; text string
data 'N' ; single character
data start_of_program ; relocatable label
```
4.15.6 PIC16 Application Example - data

This example shows the usefulness of directive `data` in storing one or more words in program memory.

```c
#include p16f877a.inc ;Include standard header file 
;for the selected device.
ORG 0x0000 ;The following code will be 
;programmed in reset address 0.
goto start ;Jump to an address labelled 
;'start'.

start ;Write your main program here.

goto $ ;Go to current line (loop here)
ORG 0x1000 ;Store the string starting from 
;1000H.

Ch_stng data 'M','C','U' ;3 program memory locations 
;will be filled with ASCII 
;equivalent of 'M','C' and 
;'U'.

Directive data produces three 14-bit numbers: 004Dh, 0043h, and 0055h. 4Dh, 43h and 55h are ASCII equivalents of 'M', 'C' and 'U', respectively.

tbl_dta data 0xffff,0xaa55 ;Places 3fffh and 2a55h in 
;two consecutive program 
;memory locations. As program 
;memory is 14-bit wide, 
;the last nibble can store 
a maximum value 3.

end
```

4.15.7 PIC18 Application Example - data

This example shows the usefulness of directive `data` in storing one or more words in program memory.

```c
#include p18f452.inc ;Include standard header file 
;for the selected device.
ORG 0x0000 ;The following code will be 
;programmed in reset address 0.
goto start ;Jump to an address labelled 
;'start'.

start ;Write your main program here.

goto $ ;Go to current line (loop here)
ORG 0x1000 ;Store the string starting from 
;1000H. In PIC18 devices, the 
;first character is in least 
;significant byte.
```
Ch_stng   data    'M','C','U'  ; 3 program memory locations
; will be filled with ASCII
; equivalent of 'M', 'C' and
; 'U'.

Directive data produces three 16-bit numbers: 004Dh, 0043h, and 0055h. 4Dh, 43h and 55h are ASCII equivalents of 'M', 'C' and 'U', respectively. See Section 4.10 "code_pack - Begin an Object File Packed Code Section (PIC18 MCUs)" for better use of memory.

Ch_stgl   data    "MCU"        ; 2 program memory locations
; will be filled with two
; words (16-bit numbers),
; each representing ASCII
; equivalent of two
; characters. The last
; character will be taken as
; NULL in case odd number of
; characters are specified.

Directive data produces two words: 434Dh and 0055h. 434Dh represents 'C' and 'M'.

tbl_dta   data   0xffff,0xaa55  ; Places ffff and aa55 in
; two consecutive program
; memory locations.

4.16  db - DECLARE DATA OF ONE BYTE

4.16.1 Syntax

[label] db expr[,expr,...,expr]

4.16.2 Description

db - Data Byte.
Reserve program memory words with 8-bit values. Multiple expressions continue to fill bytes consecutively until the end of expressions. Should there be an odd number of expressions, the last byte will be zero unless in a PIC18 code_pack section.

4.16.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 "Assembler Operation".

When generating a linkable object file, this directive can also be used to declare initialized data values. Refer to the idata directive for more information.

For PIC18 devices, use code_pack with db, since it is desired to not have bytes padded with zeroes. See the description of code_pack for more information.

4.16.4 See Also

data de dt dtm dw idata code_pack
4.16.5 Simple Examples

Example 1: PIC16 Devices

db 0x0f, 't', 0x0f, 'e', 0x0f, 's', 0x0f, 't', '
'
ASCII: 0x0F74 0x0F65 0x0F73 0x0F74 0x0a00

Example 2: PIC18 Devices

db 't', 'e', 's', 't', '
'
ASCII: 0x6574 0x7473 0x000a

4.16.6 PIC16 Application Example - db

This example shows the usefulness of directive db in storing one or more bytes or characters in program memory.

```c
#include p16f877a.inc ;Include standard header file
; for the selected device.

ORG 0x0000 ;The following code will be
; programmed in reset address 0.
goto start ;Jump to an address labelled
; 'start'.

start ;Write your main program here.
goto $ ;Go to current line (loop here)

ORG 0x1000 ;Store the string starting from
; 1000H.
Ch_stng   db 0,'M',0,'C',0,'U'

Ch_string contains three 14-bit numbers: 004Dh, 0043h, and 0055h. These are ASCII equivalents of 'M', 'C' and 'U', respectively.

tbl_dta   db 0,0xff ;Places 00ff in program memory
; location.

end
```

4.16.7 PIC18 Application Example - db

This example shows the usefulness of directive db in storing one or more byte or character in program memory.

```c
#include p18f452.inc ;Include standard header file
; for the selected device.

ORG 0x0000 ;The following code will be
; programmed in reset address 0.
goto start ;Jump to an address labelled
; 'start'.

start ;Write your main program here.
goto $ ;Go to current line (loop here)
```
ORG 0x1000 ;Store the string starting from 1000H. In PIC18 devices, the first character is in least significant byte.

Ch_stng db 'M','C','U'

Ch_stng contains three 16-bit numbers: 004Dh, 0043h, and 0055h. These are ASCII equivalents of 'M', 'C' and 'U', respectively. Information on storing data in both bytes of a program word on the PIC18 architecture can be found in Section 4.10 "code_pack - Begin an Object File Packed Code Section (PIC18 MCUs)"

tbl_dta db 0,0xff ;Places ff00 in program memory location.

end

4.17 de - DECLARE EEPROM DATA BYTE

4.17.1 Syntax

[label] de expr [, expr, ..., expr]

4.17.2 Description

de - Data EEPROM.
This directive can be used at any location for any processor.
For PIC18 devices, reserve memory word bytes are packed. If an odd number of bytes is specified, a 0 will be added unless in a code_pack section. See the description for code_pack for more information.
For all other PIC1X MCU devices, reserve memory words with 8-bit data. Each expr must evaluate to an 8-bit value. The upper bits of the program word are zeroes. Each character in a string is stored in a separate word.

4.17.3 Usage
This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.
This directive is designed mainly for initializing data in the EE data memory region of PIC1X MCU devices with EE data FLASH.
For PIC18 MCU devices, make sure to specify the start of data memory at 0xF0000. For other PIC1X MCU devices, make sure to specify the start of data memory at 0x2100. Always check your device programming specification for the correct address.

4.17.4 See Also
data db dt dtm dw code_pack

4.17.5 Simple Example
Initialize EEPROM data on a PIC16 device:

org 0x2100
dc "My Program, v1.0", 0
4.17.6  PIC16 Application Example - de

#include p16f877a.inc ;Include standard header file ;for the selected device.

org 0x2100 ;The absolute address 2100h is ;mapped to the 0000 location of ;EE data memory.

;You can create a data or character table starting from any ;address in EE data memory.

ch_tbl2 de "PICmicro" ;8 EE data memory locations ;(starting from 0) will be filled ;with 8 ASCII characters.

end

4.17.7  PIC18 Application Example - de

#include p18f452.inc ;Include standard header file ;for the selected device.

org 0xF00000 ;The absolute address F00000h is ;mapped to the 0000 location of ;EE data memory for PIC18 devices.

;You can create a data or character table starting from any ;address in EE data memory.

ch_tbl2 de "PICmicro" ;8 EE data memory locations ;(starting from 0) will be filled ;with 8 ASCII characters.

end

4.18  #define - DEFINE A TEXT SUBSTITUTION LABEL

4.18.1  Syntax

#define name [string]

4.18.2  Description

This directive defines a text substitution string. Wherever name is encountered in the assembly code, string will be substituted.

Using the directive with no string causes a definition of name to be noted internally and may be tested for using the ifdef directive.

This directive emulates the ANSI 'C' standard for #define. Symbols defined with this method are not available for viewing using MPLAB IDE.

4.18.3  Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”. 
#define is useful for defining values for constants in your program.

**Note:** A processor-specific include file exists with predefined SFR names. It is recommended that you use this file instead of defining the variables yourself. See `#include` for how to include a file in your program.

This directive is also useful with the `ifdef` and `ifndef` directives, which look for the presence of an item in the symbol table.

### 4.18.4 See Also

`#undefine` `#include` `ifdef` `ifndef`

### 4.18.5 Simple Example

```assembly
#define length 20
#define control 0x19,7
#define position(X,Y,Z) (Y-(2 * Z +X))

test_label dw position(1, length, 512)
bsf control   ; set bit 7 in f19
```

### 4.18.6 Application Example - `#define`/#`undefine`

This example shows the usage of `#define` and `#undefine` directives. A symbol name previously defined with the `#define` directive, is removed from the symbol table if `#undefine` directive is used. The same symbol may be redefined again.

```assembly
#include p16f877a.inc   ;Include standard header file 
;for the selected device.

area set 0            ;The label 'area' is assigned 
                      ;the value 0.
#define lngth  50H      ;Label 'lngth' is assigned 
                      ;the value 50H.
#define wdth   25H      ;Label 'wdth' is assigned 
                      ;the value 25H
area set lngth*wdth   ;Reassignment of label 'area'.  
                      ;So 'area' will be reassigned a 
                      ;value equal to 50H*25H.

#define lngth  0     ;Define label 'lngth' to '0'.
end

#undefine lngth        ;Undefine label 'lngth'.
#undefine wdth         ;Undefine label 'wdth'
#define lngth 0          ;Define label 'lngth' to '0'.

By using the above directives, lngth will be reassigned a value '0' and wdth will be removed from the symbol list in the list (.lst) file. The label lngth must be undefined before it can be defined as '0'.```
4.19  **dt - DEFINE TABLE (PIC12/16 MCUs)**

4.19.1  **Syntax**

[label] dt expr [, expr, ..., expr]

4.19.2  **Description**

**dt** - Define data Table.

Generates a series of \texttt{RETLW} instructions, one instruction for each \texttt{expr}. Each \texttt{expr} must be an 8-bit value. Each character in a string is stored in its own \texttt{RETLW} instruction.

4.19.3  **Usage**

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see \textbf{Section 1.6 “Assembler Operation”}. This directive is used when generating a table of data for the PIC12/16 device family. If you are using a PIC18 device, it is recommended that you use the table read/write (\texttt{TBLRD}/\texttt{TBLWT}) features. See the device data sheet for more information.

4.19.4  **See Also**

\texttt{data db de dtm dw}

4.19.5  **Simple Example**

```plaintext
dt "A Message", 0
dt FirstValue, SecondValue, EndOfValues
```

4.20  **dtm - DEFINE TABLE (EXTENDED PIC16 MCUS ONLY)**

4.20.1  **Syntax**

[label] dtm expr [, expr, ..., expr]

4.20.2  **Description**

**dtm** - Define data Table using \texttt{MOVLW}.

Generates a series of \texttt{MOVLW} instructions, one instruction for each \texttt{expr}. Each \texttt{expr} must be an 8-bit value. Each character in a string is stored in its own \texttt{MOVLW} instruction.

4.20.3  **Usage**

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see \textbf{Section 1.6 “Assembler Operation”}. This directive is used when generating a table of data for the PIC16 extended device family.

4.20.4  **See Also**

\texttt{data db de dt dw}

4.20.5  **Simple Example**

```plaintext
dtm "A Message", 0
dtm FirstValue, SecondValue, EndOfValues
```
4.21 dw - DECLARE DATA OF ONE WORD

4.21.1 Syntax

[label] dw expr[,expr,...,expr]

4.21.2 Description

dw - Data Word.

Reserve program memory words for data, initializing that space to specific values. For PIC18 devices, dw functions like db. Values are stored into successive memory locations and the location counter is incremented by one. Expressions may be literal strings and are stored as described in the db data directive.

4.21.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

When generating a linkable object file, this directive can also be used to declare initialized data values. Refer to the idata directive for more information.

While db is more common to use, you may use dw to store data in Flash PIC16FXXX devices, as many of these devices can read all 14 bits of a program memory word at run-time. See the PIC16F877A data sheet for examples and more information.

4.21.4 See Also
data db idata

4.21.5 Simple Example

dw 39, "diagnostic 39", 0x123
dw diagbase-1

4.22 else - BEGIN ALTERNATIVE ASSEMBLY BLOCK TO if CONDITIONAL

4.22.1 Syntax

Preferred:
else

Supported:
#else
 .else

4.22.2 Description

Used in conjunction with an if directive to provide an alternative path of assembly code should the if evaluate to false. else may be used inside a regular program block or macro.

4.22.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is not an instruction. It is used to perform conditional assembly of code.

4.22.4 See Also

endif if
4.22.5 Simple Example

if rate < 50
  incf speed, F
else
  decf speed, F
endif

4.22.6 Application Example - if/else/endif

See this example under if.

4.23 end - END PROGRAM BLOCK

4.23.1 Syntax

end

4.23.2 Description

Indicates the end of the program.

4.23.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”. You will need at least one end directive in any assembly program to indicate the end of a build. In a single assembly file program, one and only one end must be used. Be careful not to include files which contain end as assembly will be prematurely stopped.

4.23.4 See Also

org

4.23.5 Simple Example

#include p18f452.inc
: ; executable code
: ;
end ; end of instructions

4.24 endc - END AN AUTOMATIC CONSTANT BLOCK

4.24.1 Syntax

endc

4.24.2 Description

endc terminates the end of a cblock list. It must be supplied to terminate the list.

4.24.3 Usage

This directive is used in the following types of code: absolute. For information on types of code, see Section 1.6 “Assembler Operation”. For every cblock directive used, there must be a corresponding endc.

4.24.4 See Also

cblock
4.24.5 Examples
See the examples under cblock.

4.25 endif - END CONDITIONAL ASSEMBLY BLOCK

4.25.1 Syntax
Preferred:
endif
Supported:
#endif
.endif
.fi

4.25.2 Description
This directive marks the end of a conditional assembly block. endif may be used inside a regular program block or macro.

4.25.3 Usage
This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”. For every if directive used, there must be a corresponding endif. if and endif are not instructions, but used for code assembly only.

4.25.4 See Also
else if

4.25.5 Examples
See the examples under if.

4.26 endm - END A MACRO DEFINITION

4.26.1 Syntax
endm

4.26.2 Description
Terminates a macro definition begun with macro.

4.26.3 Usage
This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”. For every macro directive used, there must be a corresponding endm.

4.26.4 See Also
macro exitm

4.26.5 Simple Example
make_table macro arg1, arg2
  dw arg1, 0 ; null terminate table name
  res arg2 ; reserve storage
  endm
4.26.6 Application Example - macro/endm

See this example under macro.

4.27 endw - END A while LOOP

4.27.1 Syntax

Preferred:
endw

Supported:
.endw

4.27.2 Description
endw terminates a while loop. As long as the condition specified by the while directive remains true, the source code between the while directive and the endw directive will be repeatedly expanded in the assembly source code stream. This directive may be used inside a regular program block or macro.

4.27.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

For every while directive used, there must be a corresponding endw. while and endw are not instructions, but used for code assembly only.

4.27.4 See Also
while

4.27.5 Examples

See the example under while.

4.28 equ - DEFINE AN ASSEMBLER CONSTANT

4.28.1 Syntax
label equ expr

4.28.2 Description

The value of expr is assigned to label.

4.28.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

In a single assembly file program, equ is commonly used to assign a variable name to an address location in RAM. Do not use this method for assigning variables when building a linked project; use a res directive inside a data section directive (idata, udata).

4.28.4 See Also

set cblock res idata udata udata_acs udata_ovr udata_shr
4.28.5 Simple Example
four equ 4 ; assigned the numeric value of 4 to label four

4.28.6 Application Example - set/equ
See this example under set.

4.29 error - ISSUE AN ERROR MESSAGE

4.29.1 Syntax
error "text_string"

4.29.2 Description
text_string is printed in a format identical to any MPASM assembler error message.
text_string may be from 1 to 80 characters.

4.29.3 Usage
This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.
You can use this directive to generate errors for yourself or others who build your code.
You can create any error message you wish, as long as it is no longer than 80 characters.

4.29.4 See Also
messg if

4.29.5 Simple Example
error_checking macro arg1
  if arg1 >= 55 ; if arg is out of range
    error "error_checking-01 arg out of range"
  endif
endm

4.29.6 Application Example - error
This program demonstrates the error assembler directive, which sets an error message to be printed in the listing file and error file.

#include pl6f877a.inc ;Include standard header file
;for the selected device.

variable baudrate ;variable used to define
;required baud rate

baudrate set D'5600' ;Enter the required value of
;baud rate here.

if (baudrate!=D'1200')&&(baudrate!=D'2400')&&
  (baudrate!=D'4800')&&(baudrate!=D'9600')&&
  (baudrate!=D'19200')
  error "Selected baud rate is not supported"
endif

The if-endif code above outputs error if the baud rate selected is other than 1200, 2400, 4800, 9600 or 19200 Hz.
RST    CODE    0x0    ;The code section named RST
;is placed at program memory
;location 0x0. The next two
;instructions are placed in
;code section RST.

pagesel  start    ;Jumps to the location labelled
goto     start    ;’start’.

PGM    CODE    ;This is the beginning of the
;code section named PGM. It is
;a relocatable code section
;since no absolute address is
;given along with directive CODE.

start

goto $    ;Go to current line (loop here)
end

4.30   errorlevel - SET MESSAGE LEVEL

4.30.1   Syntax

errorlevel {0|1|2|+msgnum|-msgnum} [, ...]

4.30.2   Description

Sets the types of messages that are printed in the listing file and error file.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Affect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Messages, warnings, and errors printed</td>
</tr>
<tr>
<td>1</td>
<td>Warnings and errors printed</td>
</tr>
<tr>
<td>2</td>
<td>Errors printed</td>
</tr>
<tr>
<td>-msgnum</td>
<td>Inhibits printing of message msgnum</td>
</tr>
<tr>
<td>+msgnum</td>
<td>Enables printing of message msgnum</td>
</tr>
</tbody>
</table>

4.30.3   Usage

This directive is used in the following types of code: absolute or relocatable. For
information on types of code, see Section 1.6 “Assembler Operation”.

Errors cannot be disabled. Warnings may be disabled using setting 2. Messages may
be disabled using settings 1 or 2. Also, messages may be disabled individually.
However, the setting of 0, 1, or 2 overrides individual message disabling or enabling.

Be careful about disabling warnings and messages, as this can make debugging of
your code more difficult.

The most common usage for this directive is to suppress “MESSAGE 302 - Operand
Not in bank 0, check to ensure bank bits are correct”. See the Simple Example for how
to do this.

4.30.4   See Also

list error
4.30.5  Simple Example

errorlevel -302 ; Turn off banking message
  ; known tested (good) code
:
errorlevel +302 ; Enable banking message
  ; untested code
:
end

4.30.6  Application Example - errorlevel

This program demonstrates the errorlevel assembler directive, which sets the type of messages that are printed in the listing file and error file.

#include p16f877a.inc ; Include standard header file
  ; for the selected device.

errorlevel 0            ; Display/print messages,
  ; warnings and errors.

messg "CAUTION: This program has errors" ; display on build

This message will display/print for error level 0.

errorlevel 1            ; Display/print only warnings
  ; and errors.

messg "CAUTION: This program has errors" ; display message

This message will NOT display/print for error level 1 or 2.

group1  udata  0x20
  group1_var1  res  1    ; Label of this directive is not
                        ; at column 1. This will generate
                        ; a warning number 207.

Warning #207 will display/print for error level 0 or 1.

errorlevel -207       ; This disables warning whose
                      ; number is 207.

group1_var2  res 1     ; label of this directive is also
                      ; not at column 1, but no warning
                      ; is displayed/printed.

errorlevel +207       ; This enables warning whose
                      ; number is 207

group2  udata

errorlevel 2            ; Display/print only errors

group2_var1 res 1     ; label of this directive is not
                      ; at column 1. This will generate
                      ; a warning number 207.
Warning #207 will NOT display/print for error level 2.

```plaintext
errorlevel 1 ;Display/print warnings
;and errors.

group2_var2 res 1 ;label of this directive is not
;at column 1. This will generate
;a warning number 207.

RST CODE 0x0 ;The code section named RST
;is placed at program memory
;location 0x0. The next two
;instructions are placed in
;code section RST.

pagesel start ;Jumps to the location labelled
goto start ;'start'.

INTRT CODE 0x4 ;The code section named INTRT is
;placed at 0x4. The next two
;instructions are placed in
;code section INTRT

pagesel service_int ;Label 'service_int' is not
goto service_int ;defined. Hence this generates
;error[113].
```

Error 113 will always display/print, regardless of error level.

```plaintext
PGM CODE ;This is the beginning of the code
;section named 'PGM'. It is a
;relocatable code section since
;no absolute address is given along
;with directive CODE.

start
    movwf group1_var1
    goto $ ;Go to current line (loop here)
end
```

### 4.31 `exitm` - EXIT FROM A MACRO

#### 4.31.1 Syntax

`exitm`

#### 4.31.2 Description

Force immediate return from macro expansion during assembly. The effect is the same as if an `endm` directive had been encountered.

#### 4.31.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

Use this directive to prematurely end a macro, usually for a specific condition. This is similar to the C language command `break`.

#### 4.31.4 See Also

`endm macro`
4.31.5 Simple Example

```assembly
test macro filereg
    if filereg == 1 ; check for valid file
        exitm
    else
        error "bad file assignment"
    endif
endm
```

4.31.6 Application Example - `exitm`

This program demonstrates the `exitm` assembler directive, which causes an immediate exit from a macro. It is used in the example to exit from the macro when certain conditions are met.

```assembly
#include p16f877a.inc ;Include standard header file
;for the selected device.

result equ 0x20 ;Assign value 20H to label
;result.

RST CODE 0x0 ;The code section named RST
;is placed at program memory
;location 0x0. The next two
;instructions are placed in
;code section RST.

pagesel start
    goto start
;Jumps to the location labelled
;'start'.

add MACRO num1,num2
    ;'add' is a macro. The values of
;'num1' and 'num2' must be passed
;to this macro.

    if num1>0xff
        exitm
    ;force immediate return from
    ;macro during assembly.
    else
        if num2>0xff
            exitm
        ;force immediate return from
        ;macro during assembly.
        else
            movlw num1
            ;Load W register with a literal
            ;value assigned to the label
            ;'num1'.

            movwf result
            ;Load W register to an address
            ;location assigned to the label
            ;'result'.

            movlw num2
            ;Load W register with a literal
            ;value assigned to the label
            ;'num2'.

            addwf result
            ;Add W register with the memory
            ;location addressed by 'result'
            ;and load the result back to
            ;'result'.

            endif
        endif
    endif
```

```assembly
endif
```
Directives

```
endm                   ;End of 'add' MACRO

org    0010            ;My main program starts at 10H.

start                    ;The label 'start' is assigned an
                        ;address 10H.

add  .100,.256         ;Call 'add' MACRO with decimal
                        ;numbers 100 and 256 assigned to
                        ;'num1' and 'num2' labels,
                        ;respectively. EXTIM directive in
                        ;macro will force return.
                        ;Remember '.' means decimal, not
                        ;floating point.

end
```

4.32  expand - EXPAND MACRO LISTING

4.32.1  Syntax

```
expand
```

4.32.2  Description

Expand all macros in the listing file. (This is the default behavior.) This directive is
roughly equivalent to the \(/m MPASM assembler command line option, but may be
disabled by the occurrence of a subsequent noexpand.

4.32.3  Usage

This directive is used in the following types of code: absolute. For information on types
of code, see Section 1.6 “Assembler Operation”.

This directive may be useful when exploring a small range of code with many macros
in it.

4.32.4  See Also

```
macro noexpand
```

4.32.5  Simple Example

Code example:

```
;Define a macro to add two numbers
add  macro  num1,num2
    movlw  num1
    movwf  result
    movlw  num2
    addwf  result
endm

;Use macro add
add  .100,.90
```
4.33 **extern** - DECLARE AN EXTERNALLY DEFINED LABEL

4.33.1 Syntax

`extern label [, label...]`

4.33.2 Description

This directive declares symbol names that may be used in the current module but are defined as global in a different module.

The `extern` statement must be included before the `label` is used. At least one label must be specified on the line. If `label` is defined in the current module, MPASM assembler will generate a duplicate label error.

4.33.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

As soon as you have more than one file in your project, you may use this directive. `extern` will be used in a file when a label (usually a variable) is used by that file. `global` will be used in another file so that the label may be seen by other files. You must use both directives as specified or the label will not be visible to other files.

4.33.4 See Also

global idata udata udata_acs udata_oqr udata_shr

4.33.5 Simple Example

```asm
;extern Function
;
call Function
```

4.33.6 Application Example - `extern/global`

The program `main.asm`, along with `sub.asm`, demonstrate the `global` and `extern` directives, which make it possible to use symbols in modules other than where they are defined. This allows a project to be split up into multiple files (two in this example) for code reuse.

```asm
;*******************************************************
;main.asm
;*******************************************************
#include p16f877a.inc  ;Include standard header file
;for the selected device.

UDATA
delay_value  res 1

GLOBAL delay_value    ;The variable 'delay_value',
```

 declared GLOBAL in this
EXTERN delay

; The variable 'delay', declared
; EXTERN in this module, is
; declared GLOBAL in the module
; sub.asm.

RST CODE 0x0

; The code section named RST
; is placed at program memory
; location 0x0. The next two
; instructions are placed in
; code section RST.

pagesel start
go to start

; Jumps to the location labelled
; 'start'.

PGM CODE

; This is the beginning of the
; code section named PGM. It is
; a relocatable code section
; since no absolute address is
; given along with directive CODE.

start
    movlw D'10'
    movwf delay_value
    xorlw 0x80
    call delay
    goto start
end

PGM CODE

delay
    decfsz delay_value,1
    goto delay
    return
end

GLOBAL delay

; The variable 'delay' declared
; GLOBAL in this module is
; included in an EXTERN directive
; in the module main.asm.

EXTERN delay_value

; The variable 'delay_value'
; declared EXTERN in this module
; is declared GLOBAL in the
; module main.asm.
4.34 fill - SPECIFY PROGRAM MEMORY FILL VALUE

4.34.1 Syntax

[label] fill expr,count

4.34.2 Description

Generates count occurrences of the program word or byte (PIC18 devices), expr. If bounded by parentheses, expr can be an assembler instruction.

4.34.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

Note: For relocatable code, do not use a symbol in another section in the expression expr.

This directive is often used to force known data into unused program memory. This helps ensure that if code ever branches to an unused area at run-time, a fail-safe condition occurs. For example, it is not uncommon to see this used with the watchdog timer (WDT) on a PIC16 device. Unused program memory would be filled with goto or branch instructions to prevent execution of the clrwdt instruction in code, which would cause the device to reset. See the device data sheet for more information on the WDT.

4.34.4 See Also

data dw org

4.34.5 Simple Examples

Example 1: PIC10/12/16 MCU’s

fill 0x1009, 5 ; fill with a constant
fill (GOTO RESET_VECTOR), NEXT_BLOCK-$

Example 2: PIC18 Devices

#include p18f252.inc

org 0x12
failsafe goto $

org 0x100
fill (goto failsafe), (0x8000-$)/2 ;Divide by 2 for ;2-word instructions
end

4.34.6 PIC16 Application Example - fill

The fill directive is used to specify successive program memory locations with a constant or an assembly instruction.

#include p16f877a.inc ;Include standard header file ;for the selected device.

RST CODE 0x0000 ;The code section named RST ;is placed at program memory ;location 0x0. The next two ;instructions are placed in ;code section RST.
Directives

The fill directive is used to specify successive program memory locations with a constant or an assembly instruction. For PIC18 devices, only an even number is allowed to be specified as a count of locations to be filled.

```
#include p18f452.inc  ;Include standard header file
;for the selected device.

RST     CODE  0x0000  ;The code section named RST
;is placed at program memory
;location 0x0. The instruction
;'goto start' is placed in
;code section RST.

goto start  ;Jumps to the location labelled
;'start'.

fill 0, HI_INT-$  ;Fills 0 in 2 program memory
;locations: 0004 and 0006 -
;HI_INT addr. minus current addr.

HI_INT  CODE  0x0008

goto INTR_H

goto start

fill (goto start),6  ;Fills 6 locations (each location
;is 2 bytes wide) with 3 numbers
;of 2 word wide instructions
;<goto start>

LO_INT  CODE  0x0018

goto INTR_L

fill 10a9, start-$  ;Fills address 1Ch and 1Eh with
;10a9h

CODE  0x0020

start  ;Write your main program here
```

4.34.7 PIC18 Application Example - fill

The fill directive is used to specify successive program memory locations with a constant or an assembly instruction. For PIC18 devices, only an even number is allowed to be specified as a count of locations to be filled.
fill (nop), 4 ;Fills 2 locations (4 bytes) with NOP
goto $ ;Go to current line (loop here)

INTR_H
retfie

INTR_L
retfie

des

4.35  global - EXPORT A LABEL

4.35.1  Syntax

global label [, label...]

4.35.2  Description

This directive declares symbol names that are defined in the current module and should be available to other modules. At least one label must be specified on the line.

4.35.3  Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

When your project uses more than one file, you will be generating linkable object code. When this happens, you may use the global and extern directives.

global is used to make a label visible to other files. extern must be used in the file that uses the label to make it visible in that file.

4.35.4  See Also

extern idata udata udata_acs udata_ovr udata_shr

4.35.5  Simple Example

        global Var1, Var2
        global AddThree

        udata
        Var1 res 1
        Var2 res 1
        code
        AddThree
        addlw 3
        return

4.35.6  Application Example - extern/global

See this example under extern.

4.36  idata - BEGIN AN OBJECT FILE INITIALIZED DATA SECTION

4.36.1  Syntax

[label] idata [RAM_address]
4.36.2 Description

This directive declares the beginning of a section of initialized data. If label is not specified, the section is named .idata. The starting address is initialized to the specified address or will be assigned at link time if no address is specified. No code can be placed by the user in this segment.

The linker will generate a look-up table entry for each byte specified in an idata section. You must then link or include the appropriate initialization code. Examples of initialization code that may be used and modified as needed may be found with MPLINK linker sample application examples.

| Note: | This directive is not available for 12-bit instruction width (PIC10, some PIC12/PIC16) devices. |

The res, db and dw directives may be used to reserve space for variables. res will generate an initial value of zero. db will initialize successive bytes of RAM. dw will initialize successive bytes of RAM, one word at a time, in low-byte/high-byte order.

4.36.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

Use this directive to initialize your variables, or use a udata directive and then initialize your variables with values in code. It is recommended that you always initialize your variables. Relying on RAM initialization can cause problems, especially when using an emulator, as behavioral differences between the emulator and the actual part may occur.

4.36.4 See Also

extern global udata udata_acs udata_ovr udata_shr

4.36.5 Simple Example

idata
LimitL dw 0
LimitH dw D'300'
Gain   dw D'5'
Flags  db 0
String db 'Hi there!'

4.36.6 Application Example - idata

This directive reserves RAM locations for variables and directs the linker to generate a lookup table that may be used to initialize the variables specified in this section. The Starting Address of the lookup table can be obtained from the Map (.map) file. If you don’t specify a value in the idata section, the variables will be initialized with 0.

```
#include p16f877a.inc ;Include standard header file
 ;for the selected device.

group1 IDATA 0x20 ;Initialized data at location
 ;20h.

group1_var1 res 1 ;group1_var1 located at 0x20,
 ;initialized with 0.

group1_var2 res 1 ;group1_var2 located at 0x21,
 ;initialized with 0.
```

Note:
This directive is not available for 12-bit instruction width (PIC10, some PIC12/PIC16) devices.
group2 IDATA ;Declaration of group2 data. The 
;addresses for variables under 
;this data section are allocated 
;automatically by the linker.

  group2_var1  db  1,2,3,4  ;4 bytes in RAM are reserved.
  group2_var2  dw  0x1234   ;1 word in RAM is reserved.

RST  CODE  0x0  ;The code section named RST 
;is placed at program memory 
;location 0x0. The next two 
;instructions are placed in 
;code section RST.

  pagesel  start
  goto     start   ;Jumps to the location labelled
                  ;'start'.

PGM  CODE  ;This is the beginning of the 
;code section named PGM. It is 
a relocatable code section 
since no absolute address is 
given along with directive CODE.

  start
  goto $ ;Go to current line (loop here)
  end

4.37  idata_acs - BEGIN AN OBJECT FILE INITIALIZED DATA SECTION IN 
ACCESS RAM (PIC18 MCUs)

4.37.1  Syntax

[label] idata_acs [RAM_address]

4.37.2  Description

This directive declares the beginning of a section of initialized data in Access RAM. If 
label is not specified, the section is named .idata_acs. The starting address is 
initialized to the specified address or will be assigned at link time if no address is 
specified. No code can be placed by the user in this segment.

The linker will generate a look-up table entry for each byte specified in an idata section. 
You must then link or include the appropriate initialization code. Examples of 
initialization code that may be used and modified as needed may be found with 
MPLINK linker sample application examples.

The res, db and dw directives may be used to reserve space for variables. res will 
generate an initial value of zero. db will initialize successive bytes of RAM. dw will 
initialize successive bytes of RAM, one word at a time, in low-byte/high-byte order.

4.37.3  Usage

This directive is used in the following types of code: relocatable. For information on 
types of code, see Section 1.6 “Assembler Operation”.

Use this directive to initialize your variables, or use a udata directive and then initialize 
your variables with values in code. It is recommended that you always initialize your 
variables. Relying on RAM initialization can cause problems, especially when using an 
emulator, as behavioral differences between the emulator and the actual part may 
occur.

4.37.4  See Also

extern  global  udata  udata_acs  udata_ovr  udata_shr
4.37.5 Simple Example

idata_acs
LimitL dw 0
LimitH dw D'300'
Gain   dw D'5'
Flags  db 0
String db 'Hi there!'

4.38 __idlocs - SET PROCESSOR ID LOCATIONS

Note: __idlocs is preceded by two underline characters.

4.38.1 Syntax

__idlocs expr
__idlocs addr, expr (PIC18 Only)

4.38.2 Description

For PIC12 and PIC16 devices, __idlocs sets the four ID locations to the hexadecimal value of expr. For example, if expr evaluates to 1AF, the first (lowest address) ID location is zero, the second is one, the third is ten, and the fourth is fifteen.

For PIC18 devices, __idlocs sets the two-byte device ID at location addr to the hexadecimal value of expr.

Before this directive is used, the processor must be declared through the command line, the list directive, or the processor directive.

4.38.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is not commonly used, but does provide an easy method of serializing devices. __idlocs can be read by a programmer. PIC18 devices can read this value at run-time, but PIC12/16 devices cannot.

4.38.4 See Also

__config config list processor

4.38.5 Simple Example

Example 1: PIC16 Devices

#include p16f877a.inc ;Include standard header file
;for the selected device.
__idlocs 0x1234 ;Sets device ID to 1234.

Example 2: PIC18 Devices

Note: The most significant nibble of __idlocs is always 0x0, according to the programming specification.

#include p18f452.inc ;Include standard header file
;for the selected device.
__idlocs    _IDLOC0, 0x1  ;IDLOC register 0 will be
;programmed to 1.
__idlocs    _IDLOC1, 0x2  ;IDLOC register 1 will be
;programmed to 2.
__idlocs    _IDLOC2, 0x3  ;IDLOC register 2 will be
;programmed to 3.
__idlocs    _IDLOC3, 0x4  ;IDLOC register 3 will be
;programmed to 4.
__idlocs    _IDLOC4, 0x5  ;IDLOC register 4 will be
;programmed to 5.
__idlocs    _IDLOC5, 0x6  ;IDLOC register 5 will be
;programmed to 6.
__idlocs    _IDLOC6, 0x7  ;IDLOC register 6 will be
;programmed to 7.
__idlocs    _IDLOC7, 0x8  ;IDLOC register 7 will be
;programmed to 8.

4.39  if - BEGIN CONDITIONALLY ASSEMBLED CODE BLOCK

4.39.1  Syntax

Preferred:
if expr

Supported:
#if expr
.if expr

4.39.2  Description

Begin execution of a conditional assembly block. If expr evaluates to true, the code
immediately following the if will assemble. Otherwise, subsequent code is skipped until
an else directive or an endif directive is encountered.

An expression that evaluates to zero is considered logically FALSE. An expression that
evaluates to any other value is considered logically TRUE. The if and while
directives operate on the logical value of an expression. A relational TRUE expression
is guaranteed to return a nonzero value, FALSE a value of zero.

if's may be nested up to 16 deep.

4.39.3  Usage

This directive is used in the following types of code: absolute or relocatable. For
information on types of code, see Section 1.6 “Assembler Operation”.

This directive is not an instruction, but used to control how code is assembled, not how
it behaves at run-time. Use this directive for conditional assembly or to check for a
condition, such as to generate an error message.

4.39.4  See Also

else endif
4.39.5 Simple Example

if version == 100; check current version
    movlw 0x0a
    movwf io_1
else
    movlw 0x01a
    movwf io_2
endif

4.39.6 Application Example - if/else/endif

This program demonstrates the utility of if, else and endif assembly directives.

#include p16f877a.inc ;Include standard header file ;for the selected device.

variable cfab ;variable used to define ;required configuration of ;PORTA & PORTB
cfab set .1 ;Set config to decimal .1

RST CODE 0x0 ;The code section named RST ;is placed at program memory ;location 0x0. The next two ;instructions are placed in ;code section RST.

    pagesel start ;Jumps to the location labelled ;'start'.
    goto start

PGM CODE ;This is the beginning of the ;code section named PGM. It is ;a relocatable code section ;since no absolute address is ;given along with directive CODE.

    start
    banksel TRISA
    if cfab==0x0 ;If config==0x0 is true,
        clrw ;assemble the mnemonics up to
        movwf TRISA ;the directive 'else'. Set up PORTA
        movlw 0xff
        movwf TRISB ;as output.
    else
        clrw ;If config==0x0 is false,
        movwf TRISB ;assemble the mnemonics up to
        movlw 0xff
        movwf TRISA ;the directive 'endif'. Set up PORTB ;as output.
    endif
    goto $ ;Go to current line (loop here)
end
4.40 ifdef - EXECUTE IF SYMBOL HAS BEEN DEFINED

4.40.1 Syntax

Preferred:
ifdef label

Supported:
#ifdef label

4.40.2 Description

If label has been previously defined, usually by issuing a #define directive or by setting the value on the MPASM assembler command line, the conditional path is taken. Assembly will continue until a matching else or endif directive is encountered.

4.40.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 "Assembler Operation".

This directive is not an instruction, but used to control how code is assembled, not how it behaves at run-time. Use this directive for removing or adding code during debugging, without the need to comment out large blocks of code.

4.40.4 See Also

#define #undefine else endif ifndef

4.40.5 Simple Example

#define testing 1     ; set testing "on"
ifdef testing
<execute test code> ; this path would be executed.
endif

4.40.6 Application Example - ifdef

#include pl6f877a.inc
#define AlternateASM ;Comment out with ; if extra
;features not desired.
#ifdef AlternateASM
MyPort equ PORTC ;Use Port C if AlternateASM defined.
MyTris equ TRISC ;TRISC must be used to set data
;direction for PORTC.
#else
MyPort equ PORTB ;Use Port B if AlternateASM not defined.
MyTris equ TRISB ;TRISB must be used to set data
;direction for PORTB.
#endif

banksel MyTris
clrff MyTris ;Set port to all outputs.
banksel MyPort ;Return to bank used for port.
movlw 55h ;Move arbitrary value to W reg.
movwf MyPort ;Load port selected with 55h.
end
4.40.7  Application Example 2 - ifdef

This program uses the control directive #define, along with the ifdef, else and endif directives to selectively assemble code for use with either an emulator or an actual part. The control directive #define is used to create a “flag” to indicate how to assemble the code - for the emulator or for the actual device.

```
#include p18f452.inc
#define EMULATED ;Comment out with ; if actual part.

INIT
#ifdef EMULATED ;If emulator used, add lines of
    movlw 0xb0            ;initialization code to work around
    movwf 0xf9c           ;table read limitation.
#endif.

4.41 ifndef - EXECUTE IF SYMBOL HAS NOT BEEN DEFINED

4.41.1 Syntax

Preferred:
ifndef label

Supported:
#ifndef label

4.41.2 Description

If label has not been previously defined, or has been undefined by issuing an #undefine directive, then the code following the directive will be assembled. Assembly will be enabled or disabled until the next matching else or endif directive is encountered.

4.41.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is not an instruction, but used to control how code is assembled, not how it behaves at run-time. Use this directive for removing or adding code during debugging, without the need to comment out large blocks of code.

4.41.4 See Also

#define #undefine else endif ifdef

4.41.5 Simple Example

#define   testing1 ; set testing on
;
#undef   testing1 ; set testing off
    ifndef testing ; if not in testing mode
    ; execute this path
    endif
end ; end of source
4.41.6 Application Example - ifndef

```assembly
#include pl6f877a.inc
#define UsePORTB ;Comment out with ; to use PORTC

#ifndef UsePORTB
MyPort equ PORTC ;Use Port C if UsePORTB not defined.
MyTris equ TRISC ;TRISC must be used to set data
;direction for PORTC.
#else
MyPort equ PORTB ;Use Port B if UsePORTB defined.
MyTris equ TRISB ;TRISB must be used to set data
;direction for PORTB.
#endif

banksel MyTris
clrf MyTris ;Set port to all outputs.
banksel MyPort ;Return to bank used for port.
movlw 55h ;Move arbitrary value to W reg.
movwf MyPort ;Load port selected with 55h.
end
```

4.42 #include - INCLUDE ADDITIONAL SOURCE FILE

4.42.1 Syntax

Preferred:

```
#include include_file
#include "include_file"
#include <include_file>
```

Supported:

```
include include_file
include "include_file"
include <include_file>
```

4.42.2 Description

The specified file is read in as source code. The effect is the same as if the entire text of the included file were inserted into the file at the location of the include statement. Upon end-of-file, source code assembly will resume from the original source file. Up to 5 levels of nesting are permitted. Up to 255 include files are allowed.

If `include_file` contains any spaces, it must be enclosed in quotes or angle brackets. If a fully qualified path is specified, only that path will be searched. Otherwise, the search order is:

- current working directory
- source file directory
- MPASM assembler executable directory

4.42.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

You should use the `include` directive once to include that standard header file for your selected processor. This file contains defined register, bit and other names for a specific processor, so there is no need for you to define all of these in your code.
4.42.4 See Also
#define #undefine

4.42.5 Simple Example
#include p18f452.inc ;standard include file
#include "c:\Program Files\mydefs.inc" ;user defines

4.43 list - LISTING OPTIONS

4.43.1 Syntax
list [list_option, ..., list_option]

4.43.2 Description
Occurring on a line by itself, the list directive has the effect of turning listing output on, if it had been previously turned off. Otherwise, one of a list of options can be supplied to control the assembly process or format the listing file.

4.43.3 Usage
This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

TABLE 4-1: LIST DIRECTIVE OPTIONS

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>b=nnn</td>
<td>8</td>
<td>Set tab spaces.</td>
</tr>
<tr>
<td>c=nnn</td>
<td>132</td>
<td>Set column width.</td>
</tr>
<tr>
<td>f=format</td>
<td>INHX8M</td>
<td>Set the hex file output. format can be INHX32, INHX8M, or INHX8S. <strong>Note:</strong> Hex file format is set in MPLAB IDE (Build Options dialog.)</td>
</tr>
<tr>
<td>free</td>
<td>FIXED</td>
<td>Use free-format parser. Provided for backward compatibility.</td>
</tr>
<tr>
<td>fixed</td>
<td>FIXED</td>
<td>Use fixed-format parser.</td>
</tr>
<tr>
<td>mm={ON</td>
<td>OFF}</td>
<td>On</td>
</tr>
<tr>
<td>n=nnn</td>
<td>60</td>
<td>Set lines per page.</td>
</tr>
<tr>
<td>p=type</td>
<td>None</td>
<td>Set processor type; for example, PIC16F54. See also processor. <strong>Note:</strong> Processor type is set in MPLAB IDE (Configure&gt;Device).</td>
</tr>
<tr>
<td>pe=type</td>
<td>None</td>
<td>Set processor type and enable extended instruction set, for example; LIST pe=PIC18F4620 Only valid with processors which support the extended instruction set and the generic processor PIC18XXX. Is overridden by command-line option /y (disable extended instruction set). <strong>Note:</strong> Processor type is set in MPLAB IDE (Configure&gt;Device).</td>
</tr>
<tr>
<td>r=radix</td>
<td>hex</td>
<td>Set radix: hex, dec, oct. See also radix.</td>
</tr>
<tr>
<td>st={ON</td>
<td>OFF}</td>
<td>On</td>
</tr>
<tr>
<td>t={ON</td>
<td>OFF}</td>
<td>Off</td>
</tr>
<tr>
<td>w={0</td>
<td>1</td>
<td>2}</td>
</tr>
</tbody>
</table>
4.43.4 See Also

errorlevel expand noexpand nolist processor radix

4.43.5 Simple Example

Set the processor type to PIC18F452, the hex file output format to INHX32 and the radix to decimal.

list p=18f452, f=INHX32, r=DEC

4.44 local - DECLARE LOCAL MACRO VARIABLE

4.44.1 Syntax

Preferred:
local label[,label...]

Supported:
.local label[,label...]

4.44.2 Description

Declares that the specified data elements are to be considered in local context to the macro. label may be identical to another label declared outside the macro definition; there will be no conflict between the two.

If the macro is called recursively, each invocation will have its own local copy.

4.44.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

If you use a macro more than once and there is a label in it, you will get a “Duplicate Label” error unless you use this directive.

4.44.4 See Also

delm macro

4.44.5 Simple Example

<main code segment>

:;
:
len equ 10 ; global version
size equ 20 ; note that a local variable
; may now be created and modified
test macro size
local len, label ; local len and label
len set size ; modify local len
label res len ; reserve buffer
len set len-20 ; end macro
endm                   ; end macro

TABLE 4-1: LIST DIRECTIVE OPTIONS (CONTINUED)

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x={ON</td>
<td>OFF}</td>
<td>On</td>
</tr>
</tbody>
</table>

Note: All list options are evaluated as decimal numbers by default.
4.44.6 Application Example - local

This code demonstrates the utility of the `local` directive, which declares that the specified data elements are to be considered in local context to the macro.

```assembly
#include p16f877a.inc ;Include standard header file for the selected device.

incr equ 2 ;Assembler variable incr is set equal to 2.

add_incr macro ;Declaration of macro 'add_incr'.
local incr ;Local assembler variable 'incr'.

The same name incr is used in the main code, where its value is set to 2.

incr set 3 ;Local 'incr' is set to 3, in contrast to 'incr' value of 2 in main code.

clrw ;W register is set to zero
addlw incr ;W register is added to incr and result placed back
endm ;in w register.

RST CODE 0x0 ;The code section named RST is placed at program memory location 0x0. The next two instructions are placed in code section RST.
pagesel start ;Jumps to the location labelled 'start'.
goto start ;'start'.

PGM CODE ;This is the beginning of the code section named PGM. It is a relocatable code section since no absolute address is given along with directive CODE.

start
clairw ;W register set to zero.

addlw incr ;W register is added with the value of incr which is now equal to 2.

add_incr ;W register is added with the value of incr which is now equal to 3 (value set locally in the macro add_incr).
clairw ;W register is set to zero again.

addlw incr ;incr is added to W register and result placed in W register. incr value is again 2, not affected by the value set in the macro.

goto $ ;Go to current line (loop here)
end
```
4.45  **macro - DECLARE MACRO DEFINITION**

4.45.1  **Syntax**

```
label macro [arg, ... , arg]
```

4.45.2  **Description**

A macro is a sequence of instructions that can be inserted in the assembly source code by using a single macro call. The macro must first be defined, then it can be referred to in subsequent source code.

Arguments are read in from the source line, stored in a linked list and then counted. The maximum number of arguments would be the number of arguments that would fit on the source line, after the label and macro terms. Therefore, the maximum source line length is 200 characters.

A macro can call another macro, or may call itself recursively. The maximum number of nested macro calls is 16.

Please refer to Chapter 7. “Macro Language” for more information.

4.45.3  **Usage**

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

4.45.4  **See Also**

```
endm exitm local
```

4.45.5  **Simple Example**

```
;Define macro Read
Read macro device, buffer, count
  movlw device
  movwf ram_20
  movlw buffer   ; buffer address
  movwf ram_21
  movlw count    ; byte count
  call sys_21    ; subroutine call
endm :
;Use macro Read
Read 0x0, 0x55, 0x05
```

4.45.6  **Application Example - macro/endm**

This code demonstrates the utility of macro directive, which is used to define a macro.

```
#include p16f877a.inc ;Include standard header file
                  ;for the selected device.

result  equ  0x20 ;Assign value 20H to label
                 ;result.

ORG  0x0000 ;The following code will be placed
            ;in reset address 0.
goto  start ;Jump to an address whose label is
            ;'start'.

add  MACRO  num1,num2 ;'add' is a macro. The values of
                      ;'num1' and 'num2' must be passed
                      ;to this macro.
```
```assembly
movlw num1            ;Load W register with a literal 
                      ;value assigned to the label 
                      ;'num1'.

movwf result          ;Load W register to an address 
                      ;location assigned to the label 
                      ;'result'.

movlw num2            ;Load W register with a literal 
                      ;value assigned to the label 
                      ;'num2'.

addwf result          ;Add W register with the memory 
                      ;location addressed by 'result' 
                      ;and load the result back to 
                      ;'result'.

endm                   ;end of 'add' MACRO

ORG  0x0010            ;Main program starts at 10H.

start                    ;The label 'start' is assigned an 
                      ;address 10H.

add .100,.90          ;Call 'add' MACRO with decimal 
                      ;numbers 100 and 90 assigned to 
                      ;'num1' and 'num2' labels, 
                      ;respectively. 100 and 90 will be 
                      ;added and the result will be in 
                      ;'result'.

end                     

4.46  __maxram - DEFINE MAXIMUM RAM LOCATION

Note: maxram is preceded by two underline characters.

4.46.1  Syntax

__maxram expr

4.46.2  Description

The __maxram and __badram directives together flag accesses to unimplemented 
registers. __maxram defines the absolute maximum valid RAM address and initializes 
the map of valid RAM addresses to all addresses valid at and below expr. expr must 
be greater than or equal to the maximum page 0 RAM address and less than 1000H. 
This directive is designed for use with the __badram directive. Once the 
__maxram directive is used, strict RAM address checking is enabled, using the RAM 
map specified by __badram. 

__maxram can be used more than once in a source file. Each use redefines the 
maximum valid RAM address and resets the RAM map to all locations.
4.46.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 "Assembler Operation". This directive is not commonly used in user code, as RAM and ROM details are handled by the include files (*.inc) or linker script files (*.lkr).

4.46.4 See Also

__badram

4.46.5 Simple Example

See the examples for __badram.

4.47 __maxrom - DEFINE MAXIMUM ROM LOCATION

Note: maxrom is preceded by two underline characters.

4.47.1 Syntax

__maxrom expr

4.47.2 Description

The __maxrom and __badrom directives together flag accesses to unimplemented registers. __maxrom defines the absolute maximum valid ROM address and initializes the map of valid ROM addresses to all addresses valid at and below expr. expr must be greater than or equal to the maximum ROM address of the target device. This directive is designed for use with the __badrom directive. Once the __maxrom directive is used, strict ROM address checking is enabled, using the ROM map specified by __badrom.

__maxrom can be used more than once in a source file. Each use redefines the maximum valid ROM address and resets the ROM map to all locations.

4.47.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 "Assembler Operation". This directive is not commonly used in user code, as RAM and ROM details are handled by the include files (*.inc) or linker script files (*.lkr).

4.47.4 See Also

__badrom

4.47.5 Simple Example

See the examples for __badrom.

4.48 messg - CREATE USER DEFINED MESSAGE

4.48.1 Syntax

messg "message_text"
4.48.2 **Description**

Causes an informational message to be printed in the listing file. The message text can be up to 80 characters. Issuing a `messg` directive does not set any error return codes.

4.48.3 **Usage**

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive may be used to generate any desired message. It can be useful with conditional assembly, to verify in the assembled program which code was built.

4.48.4 **See Also**

`error`

4.48.5 **Simple Example**

```assembly
mssg_macro macro
  messg "mssg_macro-001 invoked without argument"
endm
```

4.48.6 **Application Example - `messg`**

This program demonstrates the `messg` assembler directive, which sets a message to be printed in the listing file and error file.

```assembly
#include p16f877a.inc ;Include standard header file ;for the selected device.

variable baudrate ;variable used to define ;required baud rate

baudrate set D'5600' ;Enter the required value of ;baud rate here.

if (baudrate!=D'1200')&&(baudrate!=D'2400')&&
  (baudrate!=D'4800')&&(baudrate!=D'9600')&&
  (baudrate!=D'19200')
  error "Selected baud rate is not supported"
  messg "only baud rates 1200,2400,4800,9600 & 19200 Hz "&&
    "are supported"
endif
```

The `if-endif` code outputs `error` and `messg` if the baud rate selected is other than 1200, 2400, 4800, 9600 or 19200 Hz.

```assembly
RST CODE 0x0 ;The code section named RST ;is placed at program memory ;location 0x0. The next two ;instructions are placed in ;code section RST.

  pagesel start ;Jumps to the location labelled
  goto start ;'start'.

PGM CODE ;This is the beginning of the ;code section named PGM. It is ;a relocatable code section ;since no absolute address is ;given along with directive CODE.
```
4.49  noexpand - TURN OFF MACRO EXPANSION

4.49.1 Syntax

noexpand

4.49.2 Description and Usage

Turns off macro expansion in the listing file.

This directive is used in the following types of code: absolute. For information on types of code, see Section 1.6 “Assembler Operation”.

4.49.3 See Also

expand

4.49.4 Simple Example

Code example:

;Define a macro to add two numbers
add  macro  num1,num2
    movlw  num1
    movwf  result
    movlw  num2
    addwf  result
    endm

;Use macro  add
add   .100,.90

Resulting listing file:

00029   noexpand
00030   add   .100,.90
00031

4.50  nolist - TURN OFF LISTING OUTPUT

4.50.1 Syntax

nolist

4.50.2 Description and Usage

Turn off listing file output.

This directive suppresses the information required for the listing file and source-level debugging. This will prevent the ability to debug the source code.

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

4.50.3 See Also

list
4.51 org - SET PROGRAM ORIGIN

4.51.1 Syntax

[label] org expr

4.51.2 Description

Set the program origin for subsequent code at the address defined in expr. If label is specified, it will be given the value of the expr. If no org is specified, code generation will begin at address zero.

For PIC18 devices, only even-numbered expr values are allowed.

When generating an object file, the org directive is interpreted as introducing an absolute CODE section with an internally generated name. For example:

L1: org 0x200

is interpreted as:

.scnname CODE 0x200
L1:

where .scnname is generated by the assembler, and will be distinct from every name previously generated in this context.

4.51.3 Usage

This directive is used in the following types of code: absolute. For information on types of code, see Section 1.6 “Assembler Operation”.

org is commonly used in single-file assembly programs whenever code needs to be placed at a particular location. Commonly used values are 0x0 (reset), 0x4 (PIC16 device interrupt vector), 0x8 (PIC18 device high-priority interrupt vector) and 0x18 (PIC18 device low-priority interrupt vector).

4.51.4 See Also

fill res end

4.51.5 Simple Example

int_1 org 0x20
; Vector 20 code goes here
int_2 org int_1+0x10
; Vector 30 code goes here

4.51.6 PIC16 Application Example - org

This example shows the usage of the org directive. Code generation begins at an address specified by org address. The origin of a data table also can be specified by this directive. A data table may be placed either in a program memory region or in an EE data memory region, as in case of a PIC1X device with EE data FLASH.

#include p16f877a.inc ;Include standard header file ;for the selected device.

org 0x0000 ;The following code will be ;placed in reset address 0.
goto Main ;Jump to an address whose label ;is 'Main'.

org 0x0004 ;The following code will be ;placed in interrupt address 4.
goto int_routine ;Jump to an address whose label is 'int_routine'.

org 0x0010 ;The following code section will be placed starting from address 10H.
Main ;Write your main program here.
;goto Main ;Loop back to 'Main'.
org 0x0100 ;The following code section will be placed starting from address 100H.
int_routine ;Write your interrupt service routine here.
;retfie ;Return from interrupt.

org 0x0100 ;You can create a data or character table starting from any address in program memory. In this case the address is 1000h.
ch_tbl1 da "PICwithFLASH" ;6 program memory locations (starting from 1000h) will be filled with six 14-bit packed numbers, each representing two 7-bit ASCII characters.

org 0x2100 ;The absolute address 2100h is mapped to the 0000 location of EE data memory in PIC16Fxxx. You can create a data or character table starting from any address in EE data memory.
ch_tbl2 de "PICwithFLASH" ;12 EE data memory locations (starting from 0) will be filled with 12 ASCII characters.
end

4.51.7 PIC18 Application Example - org

This example shows the usage of the org directive. Code generation begins at an address specified by org address. The origin of a data table also can be specified by this directive. A data table may be placed either in a program memory region or in an EE data memory region, as in case of a PIC1X device with EE data FLASH.

#include p18f452.inc ;Include standard header file for the selected device.

org 0x0000 ;The following code will be programmed in reset address 0.
goto Main ;Jump to an address whose label is
; 'Main'.
org 0x0008 ;The following code will be
;programmed in high priority
; interrupt address 8.
goto int_hi ;Jump to an address whose label is
; 'int_hi'.
org 0x0018 ;The following code will be
;programmed in low priority
; interrupt address 18h.
goto int_lo ;Jump to an address whose label is
; 'int_lo'.
org 0x0020 ;The following code section will
;be programmed starting from
;address 20H.
Main
; ;Write your main program here.
;
; goto Main ;Loop back to 'Main'
org 0x0100 ;The following code section will
;be programmed starting from
;address 100H.
int_hi
;
; ;Write your high priority
; ;interrupt service routine here.
retfie ;Return from interrupt.
org 0x0200 ;The following code section will
;be programmed starting from
;address 200H.
int_lo
;
; ;Write your low priority
; ;interrupt service routine here.
retfie ;Return from interrupt.
org 0x1000 ;You can create a data or
;character table starting from any
;address in program memory. In
;this case the address is 1000h.

ch_tbl1 db "PICwithFLASH"
end

4.52 page - INSERT LISTING PAGE EJECT

4.52.1 Syntax

page
4.52.2 Description and Usage

Inserts a page eject into the listing file.
This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

4.52.3 See Also

list subtitle title

4.53 pagesel - GENERATE PAGE SELECTING CODE (PIC10/12/16 MCUs)

4.53.1 Syntax

pagesel label

4.53.2 Description

An instruction to the linker to generate page selecting code to set the page bits to the page containing the designated label. Only one label should be specified. No operations can be performed on label. label must have been previously defined.

The linker will generate the appropriate page selecting code:
For 12-bit instruction width (PIC10F, some PIC12/PIC16) devices, the appropriate bit set/clear instructions on the STATUS register will be generated.
For 14-bit instruction width (most PIC12/PIC16) devices, a combination of BSF and BCF commands will be used to adjust bits 3 and 4 of the PCLATH register. For PIC16 extended devices, a movlp instruction is generated to set the page. If the device contains only one page of program memory, no code will be generated.
For PIC18 devices, this command will do nothing as these devices do not use paging.

4.53.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive saves you from having to manually code page bit changes. Also, since it automatically generates code, the code is much more portable.
If you are using relocatable code and your device has more than 2k program memory (or 0.5K for 12-bit instruction width devices), it is recommended that you use this directive, especially when code must jump between two or more code sections.
If you wish to indicate the start address of a RETLW table or a jump table for computed GOTOs, you must use the pageselw directive.

4.53.4 See Also

bankisel banksel

4.53.5 Simple Example

pagesel GotoDest
goto GotoDest
:
pagesel CallDest
call CallDest
4.53.6 Application Example - pagesel

This program demonstrates the pagesel directive, which generates the appropriate code to set/clear PCLATH bits. This allows easier use of paged memory such as found on PIC16 devices.

```
#include p16f877a.inc ;Include standard header file
 ;for the selected device.

RST     CODE     0x0     ;The code section named RST
 ;is placed at program memory
 ;location 0x0. The next two
 ;instructions are placed in
 ;code section RST.

pagesel start         ;Jumps to the location labelled
goto     start         ;’start’.

PGM0    CODE     0x500   ;The code section named PGM1 is
 ;placed at 0x500.

start
pagesel pagel_pgm     ;address bits 12 & 11 of
 ;pagel_pgm are copied to PCLATH
 ;4 & 3 respectively.

goto     pagel_pgm

PGM1    CODE     0x900   ;The code section named PGM1 is
 ;placed at 0x900. Label
 ;pagel_pgm is located in this
 ;code section.

pagel_pgm$ ;Go to current line (loop here)

end
```

4.54 pageselw - GENERATE PAGE SELECTING CODE USING WREG COMMANDS (PIC10/12/16 MCUs)

4.54.1 Syntax

```
pageselw label
```

4.54.2 Description

An instruction to the linker to generate page selecting code to set the page bits to the page containing the designated label. Only one label should be specified. No operations can be performed on label. label must have been previously defined.

The linker will generate the appropriate page selecting code. For 12-bit instruction width (PIC10F, some PIC12/PIC16) devices, the appropriate bit set/clear instructions on the STATUS register will be generated. For 14-bit instruction width (most PIC12/PIC16) devices, MOVLW and MOVWF instructions will be generated to modify the PCLATH. If the device contains only one page of program memory, no code will be generated.

For PIC18 devices, this command will do nothing as these devices do not use paging.

4.54.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive saves you from having to manually code page bit changes. Also, since it automatically generates code, the code is much more portable.
If you are using relocatable code and your device has more than 2k program memory (or 0.5K for 12-bit instruction width devices), it is recommended that you use this directive, especially when code must jump between two or more code sections.

You must use this directive instead of `pagesel` if you wish to indicate the start address of a RETLW table or a jump table for computed GOTOs. Only then will all the 5 top-most bits of the PC will be loaded with the appropriate value when an 8-bit offset is added to the PC. The 256-word boundaries will still have to be considered, as discussed in Application Note AN586.

### 4.54.4 See Also

`bankisel banksel`

### 4.54.5 Simple Example

```assembly
pageselw CommandTableStart ;Get the byte read and use it to
movlw CommandTableStart    ;index into our jump table. If
addwf Comm.RxTxByte,w      ;we crossed a 256-byte boundary,
btfsc STATUS,C                ;then increment PCLATH. Then load the
incf PCLATH,f                 ;program counter with computed goto.

CommandTableStart
  goto GetVersion          ;0x00 - Get Version
  goto GetRTSample        ;0x01 - Get Real Time sample
  goto Configure          ;0x02 - stub
  goto Go                 ;0x03 - stub
  goto ReadBuffer         ;0x04 - Read Buffer, just sends Vout
  goto AreYouThroughYet   ;0x05
  goto CommDone           ;0x06
  goto CommDone           ;0x07
```

### 4.55 processor - SET PROCESSOR TYPE

#### 4.55.1 Syntax

```
processor processor_type
```

#### 4.55.2 Description

Sets the processor type to `processor_type`.

#### 4.55.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is not generally used as the processor is set in MPLAB IDE (`Configure>Device`). If it must be set in code, use `processor` or the `list` directive option `p=` to set the processor.

#### 4.55.4 See Also

`list`

#### 4.55.5 Simple Example

```
processor 16f877a    ;Sets processor to PIC16F877A
```
4.56 radix - SPECIFY DEFAULT RADIX

4.56.1 Syntax

radix default_radix

4.56.2 Description

Sets the default radix for data expressions. The default radix is hex. Valid radix values are:

- hex - hexadecimal (base 16)
- dec - decimal (base 10)
- oct - octal (base 8)

You may also specify a radix using the list directive. For specifying the radix of constants, see Section 3.4 “Numeric Constants and Radix”.

4.56.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

For many programs, the default radix, hex, is used and there is no need to set the radix. However, if you need to change the radix in your program, you should exercise care, as all numeric values following the radix declaration will use that radix value. See the radix example for more on the implications of changing the radix.

Use the radix directive or the list directive option r= to change the radix in your code.

4.56.4 See Also

list

4.56.5 Simple Example

radix dec

4.56.6 Application Example - radix

This example shows the usage of the radix directive for data presentation. If not declared, then the default radix is in hexadecimal.

```
list r=dec ;Set the radix as decimal.
#include p16f877a.inc ;Include standard header file
                   ;for the selected device.

movlw 50H         ;50 is in hex
movlw 0x50        ;Another way of declaring 50 hex
movlw 500         ;50 is in octal
movlw 50          ;50 is not declared as hex or
                   ;octal or decimal. So by default
                   ;it is in decimal as default radix
                   ;is declared as decimal.

radix oct          ;Use 'radix' to declare default
                   ;radix as octal.

movlw 50H         ;50 is in hex.
movlw 0x50        ;Another way of declaring 50 hex.
movlw .50         ;50 is in decimal.
```
movlw  50               ;50 is not declared as hex or
;octal or decimal. So by default
;it is in octal as default radix
;is declared as octal.

radix  hex              ;Now default radix is in hex.

movlw  .50              ;50 is declared in decimal.
movlw  500              ;50 is declared in octal
movlw  50               ;50 is not declared as hex or
;octal or decimal. So by default
;it is in hex as default radix
;is declared as hex.

end

4.57  res - RESERVE MEMORY

4.57.1  Syntax

[label] res mem_units

4.57.2  Description

Causes the memory location pointer to be advanced from its current location by the
value specified in mem_units. In relocatable code (using MPLINK linker), res can be
used to reserve data storage. In non-relocatable code, label is assumed to be a
program memory address.

Address locations are defined in words for PIC12/16 MCUs, and bytes for PIC18
MCUs. For MPASM v3.30 and later, the res directive does not reserve an odd number
of locations for PIC18 MCUs in non-relocatable mode.

4.57.3  Usage

This directive is used in the following types of code: absolute or relocatable. For
information on types of code, see Section 1.6 “Assembler Operation”.

The most common usage for res is for data storage in relocatable code.

4.57.4  See Also

fill org equ cblock

4.57.5  Simple Example

buffer res 64 ; reserve 64 address locations of storage

4.57.6  Application Example - res

This example shows the advantage of res directive in developing relocatable code.
The program calculates the perimeter of a rectangle. Length and width of the rectangle
will be stored in buffers addressed by length and width. The calculated perimeter
will be stored in the double-precision buffer addressed by perimeter.

#include p18f452.inc ;Include standard header file
;for the selected device.

UDATA ;This directive allows the
;following data to be placed only
;in the data area.
Directives

perimeter res 2 ;Two locations of memory are reserved for the label 'perimeter'. Addresses of the memory locations will be allocated by the linker.

length res 1 ;One location of memory is reserved for the label 'length'. Address of the memory location will be allocated by the linker.

width res 1 ;One location of memory is reserved for the label 'width'. Address of the memory location will be allocated by the linker.

Start CODE 0x0000 ;Following code will be placed in address 0.

Here the directive code has the same effect as org. But org is used with MPASM assembler to generate absolute code and code is used with MPLINK linker to generate an object file. code is also different in that an address is not normally specified; the linker handles the allocation of space, both in program Flash and data RAM memory.

goto PER_CAL ;Jump to label PER_CAL

CODE ;CODE directive here dictates that the following lines of code will be placed in program memory, but the starting address will be decided by the linker.

PER_CAL
c1rf perimeter+1 ;Clear the high byte of the label 'perimeter'.
movf length,w ;Move the data present in the register addressed by 'length' to 'w'.
addwf width,w ;Add data in 'w' with data in the register addressed by 'width'. STATUS register carry bit C may be affected.
movwf perimeter ;Move 'w' to the perimeter low byte at address 20H. Carry bit is unaffected.
rlf perimeter+1 ;Increment register 21H if carry was generated. Also clear carry if bit was set.
rlf perimeter ;Multiply register 20H by 2. Carry bit may be affected.
rlf perimeter+1 ;Again, increment register 21H if carry was generated.

The previous two lines of code will multiply (by left-shifting one bit) the intermediate result by 2.

goto $ ;Go to current line (loop here)
end
4.58 set - DEFINE AN ASSEMBLER VARIABLE

4.58.1 Syntax

Preferred:

\[ \text{label set expr} \]

Supported:

\[ \text{label .set expr} \]

4.58.2 Description

\[ \text{label} \] is assigned the value of the valid MPASM assembler expression specified by \[ \text{expr} \]. The \text{set} directive is functionally equivalent to the \text{equ} directive except that \text{set} values may be subsequently altered by other \text{set} directives.

4.58.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 "Assembler Operation".

Because \text{set} directive values may be altered by later \text{set} directives, \text{set} is particularly useful when defining a variable in a loop (e.g., a while loop.)

4.58.4 See Also

\text{equ variable while}

4.58.5 Simple Example

area set 0
width set 0x12
length set 0x14
area set length * width
length set length + 1

4.58.6 Application Example - set/equ

This example shows the usage of the \text{set} directive, used for creating symbols which may be used in MPASM assembler expressions only. The symbols created with this directive do not occupy any physical memory location on the microcontroller.

\[
\#include \text{pl16f877a.inc} ; \text{Include standard header file for the selected device.}
\]

perimeter set 0 ; The label 'perimeter' is assigned value 0.
area set 0 ; The label 'area' is assigned value 0.

The value assigned by the \text{set} directive may be reassigned later.

\[
\text{length equ 50H} ; \text{The label 'length' is assigned the value 50H.}
\]

\[
\text{width equ 25H} ; \text{The label 'width' is assigned the value 25H.}
\]
The value assigned by the `equ` directive may **not** be reassigned later.

```asm
perimeter set 2*(lngth+wdth) ;Both 'perimeter' and
area set lngth*wdth ;'area' values are
;reassigned.
end
```

### 4.59 `space` - INSERT BLANK LISTING LINES

#### 4.59.1 Syntax

**Preferred:**

```asm
space expr
```

**Supported:**

```asm
spac expr
```

#### 4.59.2 Description and Usage

Insert `expr` number of blank lines into the listing file.

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

#### 4.59.3 See Also

`list`

#### 4.59.4 Simple Example

```asm
space 3 ;Inserts three blank lines
```

### 4.60 `subtitle` - SPECIFY PROGRAM SUBTITLE

#### 4.60.1 Syntax

**Preferred:**

```asm
subtitle "sub_text"
```

**Supported:**

```asm
stitle "sub_text"
subtitl "sub_text"
```

#### 4.60.2 Description and Usage

`sub_text` is an ASCII string enclosed in double quotes, 60 characters or less in length. This directive establishes a second program header line for use as a subtitle in the listing output.

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

#### 4.60.3 See Also

`list title`

#### 4.60.4 Simple Example

```asm
subtitle "diagnostic section"
```
4.61 title - SPECIFY PROGRAM TITLE

4.61.1 Syntax

title "title_text"

4.61.2 Description and Usage

title_text is a printable ASCII string enclosed in double quotes. It must be 60 characters or less. This directive establishes the text to be used in the top line of each page in the listing file.

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 "Assembler Operation".

4.61.3 See Also

list subtitle

4.61.4 Simple Example

title "operative code, rev 5.0"

4.62 udata - BEGIN AN OBJECT FILE UNINITIALIZED DATA SECTION

4.62.1 Syntax

[label] udata [RAM_address]

4.62.2 Description

This directive declares the beginning of a section of uninitialized data. If label is not specified, the section is named .udata. The starting address is initialized to the specified address or will be assigned at link time if no address is specified. No code can be generated in this segment. The res directive should be used to reserve space for data.

Note: Two sections in the same source file are not permitted to have the same name.

4.62.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 "Assembler Operation".

For relocatable code, this directive is used to create a data (RAM) section. For absolute code, do not use this directive. Use directives equ or cblock.

4.62.4 See Also

extern global idata udata_acs udata_ovr udata_shr

4.62.5 Simple Example

udata
Var1 res 1
Double res 2

4.62.6 Application Example - udata

This program demonstrates the udata directive, which declares the beginning of a section of uninitialized data. udata does not set (initialize) the starting value of the variables; you must do this in code.
#include p16f877a.inc ;Include standard header file ;for the selected device.

group1 udata 0x20 ;group1 data stored at locations ;starting at 0x20.
group1_var1 res 1 ;group1_var1 located at 0x20.
group1_var2 res 1 ;group1_var2 located at 0x21.

group2 udata ;Declaration of group2 data. The ;addresses for variables under ;this data section are allocated ;automatically by the linker.
group2_var1 res 1 ;this data section are allocated
group2_var2 res 1

RST CODE 0x0 ;The code section named RST ;is placed at program memory ;location 0x0. The next two ;instructions are placed in ;code section RST.
pagesel start ;Jumps to the location labelled ;'start'.
goto start ;This is the beginning of the ;code section named PGM. It is ;a relocatable code section ;since no absolute address is ;given along with directive CODE.

start
banksel group1_var1
clrf group1_var1
clrf group1_var2

banksel group2_var1
clr group2_var1
clr group2_var2
goto $ ;Go to current line (loop here)

4.63 udata_acs - BEGIN AN OBJECT FILE ACCESS UNINITIALIZED DATA SECTION (PIC18 MCUs)

4.63.1 Syntax
[label] udata_acs [RAM_address]

4.63.2 Description
This directive declares the beginning of a section of access uninitialized data. If label is not specified, the section is named .udata_acs. The starting address is initialized to the specified address or will be assigned at link time if no address is specified. This directive is used to declare variables that are allocated in access RAM of PIC18 devices. No code can be generated in this segment. The res directive should be used to reserve space for data.

Note: Two sections in the same source file are not permitted to have the same name.
4.63.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is similar to udata, except that it is used only for PIC18 devices and will only place variables in access RAM. PIC18 devices have an area of RAM known as access RAM. Variables in access memory can be used no matter where the bank select register (BSR) is pointing. It is very useful for frequently-used and global variables.

4.63.4 See Also

extern global idata udata udata_ovr udata_shr

4.63.5 Simple Example

udata_acs
Var1 res 1
Double res 2

4.63.6 Application Example - udata_acs

This program demonstrates the udata_acs directive. This directive declares the beginning of a section of uninitialized data.

#include p18f452.inc ;Include standard header file
;for the selected device.

group1 udata_acs 0x20 ;group1 data stored at access
;RAM locations starting at 0x20.
group1_var1 res 1 ;group1_var1 located at 0x20.
group1_var2 res 1 ;group1_var2 located at 0x21.
group2 udata_acs ;Declaration of group2 data. The
;addresses for data under this
;section are allocated
;automatically by the linker.
group2_var1 res 1 ;All addresses be will allocated
;in access RAM space only.
group2_var2 res 1

RST CODE 0x0 ;The code section named RST
;is placed at program memory
;location 0x0. The instruction
;'goto start' is placed in
;code section RST.
goto start ;Jumps to the location labelled
;'start'.

PGM CODE ;This is the beginning of the code
;section named PGM. It is a
;relocatable code section
;since no absolute address is
;given along with directive CODE.

start
clrf group1_var1,A ;group1_var1 initialized to zero
clrf group1_var2,A ;group1_var2 initialized to zero
clrf group2_var1,A ;group2_var1 initialized to zero
clrf group2_var2,A ;group2_var2 initialized to zero
In the code above, “A” references the access RAM. This A designation can be explicitly stated by the code, but is not needed since the assembler will automatically locate variables in access memory, if possible.

4.64 udata_ovr - BEGIN AN OBJECT FILE OVERLAID UNINITIALIZED DATA SECTION

4.64.1 Syntax

[<label>] udata_ovr [RAM_address]

4.64.2 Description

This directive declares the beginning of a section of overlaid uninitialized data. If <label> is not specified, the section is named .udata_ovr. The starting address is initialized to the specified address or will be assigned at link time if no address is specified. The space declared by this section is overlaid by all other udata_ovr sections of the same name. It is an ideal way of declaring temporary variables since it allows multiple variables to be declared at the same memory location. No code can be generated in this segment. The res directive should be used to reserve space for data.

Note: Two sections in the same source file are not permitted to have the same name.

4.64.3 Usage

This directive is used in the following types of code: relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is similar to udata, except that it allows you to reuse data space by “overlaying” one data area on another. It is used for temporary variables, as each data section may overwrite (and thus share) the same RAM address locations.

4.64.4 See Also

extern global idata udata udata_acs udata_shr

4.64.5 Simple Example

Temps udata_ovr
Temp1 res 1
Temp2 res 1
Temp3 res 1
Temps udata_ovr
LongTemp1 res 2 ; this will be a variable at the ; same location as Temp1 and Temp2
LongTemp2 res 2 ; this will be a variable at the ; same location as Temp3
4.64.6 Application Example - udata_ovr

This program demonstrates the udata_ovr directive. This directive declares the beginning of a section of overlaid uninitialized data.

```assembly
#include p16f877a.inc ;Include standard header file for the selected device.

same_var udata_ovr 0x20 ;Declares an overlaid uninitialized data section named 'same_var' starting at location 0x20.
var1 res 1

same_var udata_ovr 0x20 ;Declares an overlaid uninitialized data section with the same name as the one declared above. Thus variables var1 and var2 are allocated at the same address.
var2 res 1

RST CODE 0x0 ;The code section named RST is placed at program memory location 0x0. The next two instructions are placed in code section RST.

goto start ;'start'.

PGM CODE ;This is the beginning of the code section named PGM. It is a relocatable code section since no absolute address is given along with directive CODE.

start

banksel var1 ;Any operation on var1 affects var2 also since both variables are overlaid.

movlw 0xFF

movwf var1

comf var2

goto $ ;Go to current line (loop here)

end
```
4.65  udata_shr - BEGIN AN OBJECT FILE SHARED UNINITIALIZED DATA SECTION (PIC12/16 MCUs)

4.65.1  Syntax

[label] udata_shr [RAM_address]

4.65.2  Description

This directive declares the beginning of a section of shared uninitialized data. If label
is not specified, the section is named .udata_shr. The starting address is initialized
to the specified address or will be assigned at link time if no address is specified. This
directive is used to declare variables that are allocated in RAM that is shared across all
RAM banks (i.e. unbanked RAM). No code can be generated in this segment. The res
directive should be used to reserve space for data.

Note: Two sections in the same source file are not permitted to have the same
name.

4.65.3  Usage

This directive is used in the following types of code: relocatable. For information on
types of code, see Section 1.6 “Assembler Operation”.

This directive is similar to udata, except that it is only used on parts with memory
accessible from multiple banks. udata_shr sections are used with SHAREBANK
locations in the linker script, where as udata sections are used with DATABANK
locations in the linker script. See the data sheet for the PIC16F873A for a specific
element.

4.65.4  See Also

extern global idata udata udata_acs udata_ovr

4.65.5  Simple Example

Temps udata_shr
Temp1 res 1
Temp2 res 1
Temp3 res 1

4.65.6  Application Example - udata_shr

This program demonstrates the udata_shr directive. This directive declares the
beginning of a section of shared uninitialized data. This directive is used to declare
variables that are allocated in RAM that is shared across all RAM banks (i.e. unbanked
RAM.)

#include p16f877a.inc ;Include standard header file
;for the selected device.

shared_data udata_shr ;Declares the beginning of a data
;section named 'shared data',
var res 1 ;which is shared by all banks.
; 'var' is the location which can
;be accessed irrespective of
;banksel bits.

bank0_var udata 0X20 ;Declares beginning of a data
var0 res 1 ;section named 'bank0_var',
;which is in bank0. var0 is
;allocated the address 0x20.
bank1_var udata 0xa0 ;Declares beginning of a data
   var1 res 1 ;section named 'bank1_var',
             ;which is in bank1. var1 is
             ;allocated the address 0xa0

bank2_var udata 0x120 ;Declares beginning of a data
   var2 res 1 ;section named 'bank2_var',
             ;which is in bank2. var2 is
             ;allocated the address 0x120

bank3_var udata 0x1a0 ;Declares beginning of a data
   var3 res 1 ;section named 'bank3_var',
             ;which is in bank3. var3 is
             ;allocated the address 0x1a0

RST CODE 0x0 ;The code section named RST
             ;is placed at program memory
             ;location 0x0. The next two
             ;instructions are placed in
             ;code section RST.

pagesel start ;Jumps to the location labelled
goto start ;'start'.

PGM CODE ;This is the beginning of the
           ;code section named PGM. It is
           ;a relocatable code section
           ;since no absolute address is
           ;given along with directive CODE.

start
   banksel var0 ;Select bank0.
   movlw 0x00
   movwf var ;var is accessible from bank0.

   banksel var1 ;Select bank1.
   movlw 0x01
   movwf var ;var is accessible from bank1
             ;also.

   banksel var2 ;Select bank2.
   movlw 0x02
   movwf var ;var is accessible from bank2
             ;also.

   banksel var3 ;Select bank3.
   movlw 0x03
   movwf var ;var is accessible from bank3
             ;also.

   goto $ ;Go to current line (loop here)
end

4.66 #undefine - DELETE A SUBSTITUTION LABEL

4.66.1 Syntax

#undefine label
4.66.2 Description

label is an identifier previously defined with the #define directive. The symbol named is removed from the symbol table.

4.66.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is most often used with the ifdef and ifndef directives, which look for the presence of an item in the symbol table.

4.66.4 See Also

#define #include ifdef ifndef

4.66.5 Simple Example

#define length 20
:
#undef length

4.66.6 Application Example - #define/#undefine

See this example under #define.

4.67 variable - DECLARE SYMBOL VARIABLE

4.67.1 Syntax

variable label[=expr][,label[=expr]...]

4.67.2 Description

Creates symbols for use in MPASM assembler expressions. Variables and constants may be used interchangeably in expressions.

The variable directive creates a symbol that is functionally equivalent to those created by the set directive. The difference is that the variable directive does not require that symbols be initialized when they are declared.

The variable values cannot be updated within an operand. You must place variable assignments, increments, and decrements on separate lines.

4.67.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is most used for conditional assembly code.

Note: variable is not used to declare a run-time variable, but a variable that is used by the assembler. To create a run-time variable, refer to the directives res, equ or cblobck.

4.67.4 See Also

constant set

4.67.5 Simple Example

variable RecLength=64 ; Set Default
                   ; RecLength
constant BufLength=512 ; Init BufLength
RecLength may be reset later in RecLength=128;

constant MaxMem=RecLength+BufLength;CalcMaxMem

4.67.6 Application Example - variable/constant

This example shows the usage of the variable directive, used for creating symbols which may be used in MPASM assembler expressions only. The symbols created with this directive do not occupy any physical memory location of microcontroller.

```
#include p16f877a.inc ;Include standard header file ;for the selected device.

variable perimeter=0 ;The symbol 'perimeter' is initialized to 0
variable area ;If a symbol is declared as variable, then initialization is optional, i.e. it may or may not be initialized.

constant lngth=50H ;The symbol 'lngth' is initialized to 50H.
constant wdth=25H ;The symbol 'wdth' is initialized to 25H.
;A constant symbol always needs to be initialized.

perimeter=2*(lngth+wdth) ;The value of a CONSTANT cannot be reassigned after having been initialized once. So 'lngth' and 'wdth' cannot be reassigned. But 'perimeter' has been declared as variable, and so can be reassigned.

area=lngth*wdth
```

4.68  while - PERFORM LOOP WHILE CONDITION IS TRUE

4.68.1 Syntax

Preferred:
```
while expr :
  endw
```

Supported:
```
 .while expr :
  .endw
```

4.68.2 Description

The lines between the while and the endw are assembled as long as expr evaluates to TRUE. An expression that evaluates to zero is considered logically FALSE. An expression that evaluates to any other value is considered logically TRUE. A relational TRUE expression is guaranteed to return a non-zero value; FALSE a value of zero.
A while loop can contain at most 100 lines and be repeated a maximum of 256 times. While loops can be nested up to 8 deep.

### 4.68.3 Usage

This directive is used in the following types of code: absolute or relocatable. For information on types of code, see Section 1.6 “Assembler Operation”.

This directive is not an instruction, but used to control how code is assembled, not how it behaves at run-time. Use this directive for conditional assembly.

### 4.68.4 See Also

endw if

### 4.68.5 Simple Example

while is not executed at run-time, but produces assembly code based on a condition. View the list file (*.lst) or disassembly window to see the results of this example.

test_mac macro count
    variable i
    i = 0
    while i < count
        movlw i
        i += 1
    endw
endm

start
    test_mac 5
end

### 4.68.6 Application Example - while/endw

This example shows the usefulness of directive while to perform a loop while a certain condition is true. This directive is used with the endw directive.

```c
#include p16f877a.inc  ;Include standard header file for the selected device.

variable i  ;Define the symbol 'i' as a variable.

mydata udata 0x20  ;Allocate RAM for labels
    reg_hi res 1  ;reg_hi and reg_lo.
    reg_lo res 1

RST CODE 0x0  ;The code section named RST is placed at program memory location 0x0. The next two instructions are placed in code section RST.
    pagesel start  ;Jumps to the location labelled 'start'.
    goto start

shift_right macro by_n  ;Beginning of a macro, which shifts register data n times.
    ;Code length generated after assembly, varies depending upon the value of parameter 'by_n'.
```
i=0                       ;Initialize variable i.
while  i< by_n          ;Following 3 lines of assembly
    ;code are repeated as long as
    ;i< by_n.

Up to 100 lines of codes are allowed inside a while loop.

    bcf    STATUS,C         ;Clear carry bit.
    rrf    reg_hi,F         ;reg_hi and reg_lo contains
    rrf    reg_lo,F         ;16-bit data which is rotated
    i+=1                      ;right through carry.
    i cannot increment to more than 255 decimal.

    endw                    ;End while loop. The loop will
    endm                    ;break here after i=by_n.
PGM     CODE              ;This is the beginning of the
                      ;code section named PGM. It is
                      ;a relocatable code section
                      ;since no absolute address is
                      ;given along with directive CODE.

start
    movlw   0x88           ;Initialize reg_hi and
    movwf   reg_hi         ;reg_lo for observation.
    movlw   0x44
    movwf   reg_lo

    shift_right  3           ;Shift right 3 times the 16-bit
    ;data in reg_hi and reg_lo. This
    ;is an example. A value 8 will
    ;shift data 8 times.

    goto  $                 ;Go to current line (loop here)
end
Chapter 5. Assembler Examples, Tips and Tricks

5.1 INTRODUCTION

The usage of multiple MPASM assembler directives is shown through examples. Directives are assembler commands that appear in the source code but are not opcodes. They are used to control the assembler: its input, output, and data allocation. Many of the assembler directives have alternate names and formats. These may exist to provide backward compatibility with previous assemblers from Microchip and to be compatible with individual programming practices. If portable code is desired, it is recommended that programs be written using the specifications contained within this document.

For a reference listing of all directives discussed in examples here, please see Chapter 4. “Directives”.

Note: Although MPASM assembler is often used with MPLINK object linker, MPASM assembler directives are not supported in MPLINK linker scripts. See MPLINK object linker documentation for more information on linker options to control listing and hex file output.

Topics covered are:

• Example of Displaying Count on Ports
• Example of Port B Toggle and Delay Routines
• Example of Calculations with Variables and Constants
• Example of a 32-Bit Delay Routine
• Example of SPI Emulated in Firmware
• Example of Hexadecimal to ASCII Conversion
• Other Sources of Examples
• Tips and Tricks
5.2  EXAMPLE OF DISPLAYING COUNT ON PORTS

Directives highlighted in this example are:

- `#include`
- `end`

5.2.1  Program Functional Description

This simple program continually increases the count on PORTA and PORTB. This count may be displayed in software in the SFR or watch window of MPLAB IDE, or in hardware on connected LEDs or a scope. The count may be slowed down using a delay routine (see other examples.)

Once the count has increased to 0xFF, it will roll over to 0x00 and begin again.

The application is written as absolute code, i.e., you use only the assembler to generate the executable (not the assembler and linker).

The standard header file for the processor selected is included using `#include`. The port output data latches are then cleared. Port A must be set up for digital I/O as, on power-up, several pins are analog. Data direction registers (TRISx) are cleared to set port pins to outputs. A loop named Loop is entered where the value of each port is increased indefinitely until the program is halted. Finally, the program is finished with an end.

5.2.2  Commented Code Listing

`;Toggles Port pins with count on PIC18F8720
;PortA pins on POR:
; RA5, RA3:0 = analog inputs
; RA6, RA4 = digital inputs
;PortB pins on POR:
; RB7:0 = digital inputs

#include p18f8720.inc  ;Include file needed to reference
;data sheet names.

clrf  PORTA            ;Clear output data latches on Ports
clrf  PORTB
movlw 0x0F             ;Configure Port A for digital I/O
movwf ADCON1
clrf  TRISA            ;Set data direction of Ports as outputs
clrf  TRISB

Loop
  incf  PORTA,F         ;Read PORTA, add 1 and save back.
  incf  PORTB,F         ;Read PORTB, add 1 and save back.
goto Loop              ;Do this repeatedly - count.
end                     ;All programs must have an end directive.`
5.3  EXAMPLE OF PORT B TOGGLE AND DELAY ROUTINES

Directives highlighted in this example are:
- udata, res
- equ
- code
- banksel, pagesel

Items covered in this example are:
- Program Functional Description
- Commented Code Listing
- Header Files
- Register and Bit Assignments
- Program Memory CODE Sections and Paging
- Banking
- Interrupts

5.3.1  Program Functional Description

This program continually alternates the output on the Port B pins from 1’s to 0’s. Two delay routines using interrupts provide the timing for the alternating output. If LEDs were attached to Port B, they would flash (1=on, 0=off).

The type of PIC1X MCU is set in MPLAB IDE and so does not need to be set in code. However, if you wish to specify the MCU, as well as radix, in code, you may do so using the processor and radix directives, or list command, i.e., list p=16f877a, r=hex.

The application is written as relocatable code, i.e., you must use both the assembler and linker to generate the executable. See PIC1X MCU Language Tools and MPLAB IDE for information on how to set up a project using assembler files and a linker script.

The standard header file for the processor selected is included using #include. Registers are assigned using the udata, res and equ directives. Sections of code are created using the code statement. Data memory banking and program memory paging is accomplished as needed using banksel and pagesel directives. Finally, the program is finished with an end.

5.3.2  Commented Code Listing

;******************************************************
;* MPASM Assembler Control Directives *                *
;* Example Program 1                                   *
;* Alternate output on Port B between *                *
;* 1's and 0's                                        *
;******************************************************

#include p16f877a.inc ;Include header file

MPLAB IDE has many header files (*.inc) available for supported devices. These can be found in the installation directory. See Section 5.3.3 “Header Files” for more on headers.
udata                        ;Declare storage of RAM variables
DTEMP  res 1                 ;Reserve 1 address location
DFLAG  res 1                 ;Reserve 1 address location
DFL0   equ 0x00              ;Set flag bit - 0 bit of DFLAG

Set DTEMP to be a temporary register at a location in RAM determined at by the linker. Set DFLAG to be the flag register at a location following the DTEMP register. Set DFL0 to a value to represent a bit in the DFLAG register; in this case 0. See the Additional Comments section for more information.

rst     code     0x00       ;Reset Vector
          pagesel Start       ;Ensure correct page selected
goto    Start             ;Jump to Start code

Place the reset vector at program memory location 0x00. When the program resets, the program will branch to Start.

intrpt  code     0x04       ;Interrupt Vector
goto    ServInt           ;Jump to service interrupt

Place interrupt vector code at program memory location 0x04, since this device automatically goes to this address for interrupts. When an interrupt occurs, the program will branch to the ServInt routine.

isr     code     0x08       ;Interrupt Service Routine
          ServInt
          banksel OPTION_REG  ;Select Option Reg Bank (1)
          bsf     OPTION_REG, T0CS  ;Stop Timer0
          banksel INTCON       ;Select INTCON Bank (0)
          bcf     INTCON, T0IF  ;Clear overflow flag
          bcf     DFLAG, DFL0    ;Clear flag bit
          retfie               ;Return from interrupt

For the PIC16F877A, there is not enough memory to add a pagesel ServInt statement to insure proper paging. Therefore, the ISR code needs to be specifically placed on page 0. See Section 5.3.7 “Interrupts” for more on the ISR code.

;*****************************************************************************
;* Main Program                                                            *
;*****************************************************************************

  code                        ;Start Program

Begin program code. Because no address is specified, the program memory location will be determined by the linker. See Section 5.3.5 “Program Memory CODE Sections and Paging” for more on code.
Start
    clr PORTB ;Clear PortB
    banksel TRISB ;Select TRISB Bank (1)
    clr TRISB ;Set all PortB pins as outputs
    banksel INTCON ;Select INTCON Bank (0)
    bsf INTCON, GIE ;Enable Global Int's
    bsf INTCON, T0IE ;Enable Timer0 Int

First, set up Port B pins to be all outputs using the data direction (TRISB) register. Then set up Timer 0 and interrupts for later use.

Loop
    movlw 0xFF
    movwf PORTB ;Set PortB
    call Delay1 ;Wait
    clr PORTB ;Clear PortB
    pagesel Delay2 ;Select Delay2 Page
    call Delay2 ;Wait
    pagesel Loop ;Select Loop Page
    goto Loop ;Repeat

Set all Port B pins high and wait Delay 1. Then, set all Port B pins low and wait Delay 2. Repeat until program halted. This will have the effect of “flashing” the pins of Port B.

;******************************************
;* Delay 1 Routine - Timer0 delay loop    *
;******************************************

Delay1
    movlw 0xF0 ;Set Timer0 value
    movwf TMR0 ;0x00-longest delay
               ;0xFF-shortest delay
    clr DFLAG ;Set flag bit
    bsf DFLAG, DFL0 ;Select DFLAG Bank (0)
    banksel OPTION_REG ;Select Option Reg Bank (1)
    bcf OPTION_REG, T0CS ;Start Timer0
    banksel DFLAG ;Select DFLAG Bank (0)

TLoop ;Wait for overflow: 0xFF->0x00
    btfsc DFLAG, DFL0 ;After interrupt, DFL0 = 0
    goto TLoop
    return

Use Timer 0 to create Delay 1. First, give the timer an initial value. Then, enable the timer and wait for it to overflow from 0xFF to 0x00. This will generate an interrupt, which will end the delay. See Section 5.3.7 “Interrupts” for more information.
Place Delay2 routine at program memory location 0x1000, on page 2. (See Section 5.3.5 “Program Memory CODE Sections and Paging” for more on code.) This code was placed on a page other than 0 to demonstrate how a program functions across pages.

```
decdelay code  0x1000 ; Page 2
```

Delay2

```
movlw  0xFF            ; Set DTEMP value
movwf   DTEMP           ; 0x00 - shortest delay
 ; 0xFF - longest delay

DLoop                     ; Use a simple countdown to
   decfsz  DTEMP, F        ; create delay.
   goto    DLoop           ; End loop when DTEMP=0

return
```

Use the time it takes to decrement a register DTEMP from an initial value to 0x00 as Delay 2. This method requires no timers or interrupts.

```
end
```

End of the program, i.e., tells the assembler no further code needs to be assembled.

### 5.3.3 Header Files

A header file is included in the program flow with the `#include` directive.

```
#include p16f877a.inc      ; Include header file
```

Angle brackets, quotes or nothing at all may used to enclose the name of the header file. You may specify the complete path to the included file, or let the assembler search for it. For more on search order, see the discussion of the `#include` directive in Section 4.42 “#include - Include Additional Source File”

A header file is extremely useful for specifying often-used constants, such as register and pin names. This information can be typed in once, and then the file can be included in any code using the processor with those registers and pins.
5.3.4 Register and Bit Assignments

You can specify your own registers and bits by using the udata, res and equ directives, as is done in the following lines.

```assembly
udata ;Declare storage of RAM variables
DTEMP res 1 ;Reserve 1 address location
DFLAG res 1 ;Reserve 1 address location
DFL0 equ 0x00 ;Set flag bit - 0 bit of DFLAG
```

DTEMP and DFLAG are assigned one address location in RAM each by the linker. For illustrative purposes, suppose the locations selected by the linker are the general purpose registers (GPRs) 0x20 and 0x21. DFL0 is assigned the value 0x00 and will be used as the name for pin 0 in the DFLAG register.

**FIGURE 5-1: PIC16F877A REGISTER FILE MAP**

The directives udata and res are used in relocatable code to define multiple registers instead of equ. For more on these directives, see:

- Section 4.62 “udata - Begin an Object File Uninitialized Data Section”
- Section 4.57 “res - Reserve Memory”
- Section 4.28 “equ - Define an Assembler Constant”

5.3.5 Program Memory CODE Sections and Paging

The code directive is used to specify sections of relocatable code. For absolute code, the org directive is used. See Chapter 6. “Relocatable Objects” for more on the differences between relocatable and absolute code. For more on these directives, see:

- Section 4.9 “code - Begin an Object File Code Section”
- Section 4.51 “org - Set Program Origin”

If no code directive is used, code generation will begin at address zero. For this example, code is used to specify code at 0x00 (reset address), 0x04 (interrupt address), 0x08 (interrupt service routine) and 0x1000 (Delay 2 address). It does not explicitly set the program start address, but allows the linker to place the code appropriately. Since the linker places addressed code first, and then attempts to place the relocatable code, based on size, the likely program memory usage is shown below.
Since the actual location of the main code (.code section) is unknown, pagesel directives must be used to ensure that program branches to other sections are correct.

rst code 0x00 ;Reset Vector
pagesel Start
goto Start :
code ;Start Program :
pagesel Delay2 ;Select Delay2 Page
call Delay2 ;Wait :
pagesel Loop ;Select Loop Page
goto Loop ;Repeat :

For more on this directive, see Section 4.53 “pagesel - Generate Page Selecting Code (PIC10/12/16 MCUs)”

5.3.6 Banking

In this example, Port B must be configured, causing a switch to data memory bank 1 to access the TRISB register. This change to bank 1, and subsequent return to bank 0, is easily accomplished using the banksel directive.

banksel TRISB ;Select TRISB Bank (1)
clrf TRISB ;Set PortB as output
banksel INTCON ;Select INTCON Bank (0)
bsf INTCON, GIE ;Enable Global Int's
bsf INTCON, T0IE ;Enable Timer0 Int

Two other routines also use banksel to access the Option register (OPTION_REG). For more on this directive, see Section 4.7 “banksel - Generate Bank Selecting Code”
5.3.7 Interrupts

The Delay 1 routine in this program uses the Timer 0 overflow interrupt as a timing mechanism. Once the interrupt occurs, the program branches to the interrupt vector. Here code is located to jump to a location where interrupt-handling code is found.

```
intrpt  code  0x04       ;Interrupt Vector
    goto    ServInt           ;Jump to service interrupt
```

The interrupt-handling code, also known as the interrupt service routine or ISR, is generated by the programmer to handle the specific requirements of the peripheral interrupt and the program. In this case, Timer 0 is stopped and its flag bit is cleared, so it may be run again. Then, the program-defined flag bit is cleared. Finally, `retfie` takes the program back to the instruction that was about to be executed when the interrupt occurred.

```
isr     code  0x08       ;Interrupt Service Routine
    ServInt
    banksel OPTION_REG        ;Select Option Reg Bank (1)
    bsf     OPTION_REG, T0CS  ;Stop Timer0
    banksel INTCON            ;Select INTCON Bank (0)
    bcf     INTCON, T0IF      ;Clear overflow flag
    bcf     DFLAG, DFL0       ;Clear flag bit
    retfie                    ;Return from interrupt
```

When the program code begins to execute again, the cleared flag bit `DFL0` now causes the delay loop `TLOOP` to end, thus ending Delay 1 routine.
5.4  EXAMPLE OF CALCULATIONS WITH VARIABLES AND CONSTANTS

Directives highlighted in this example are:
  • #define, #undefine
  • set
  • constant, variable

Items covered in this example are:
  • Program Functional Description
  • Commented Code Listing
  • Using Watch Windows

5.4.1  Program Functional Description

This program performs several calculations using defined constants and variables. The application is written as relocatable code, i.e., you must use both the assembler and linker to generate the executable.

The standard header file for the processor selected is included using #include.

Sections of code are created using the code statement.

5.4.2  Commented Code Listing

;**************************************
;* MPASM Assembler Control Directives *
;* Example Program 2                  *
;* Perform calculations               *
;**************************************
#include p16f877a.inc               ;Include header file
#define    Tdistance1  50           ;Define the symbol
#define    Tdistance2  25           ;Define the symbol
#undefine  Tdistance2               ;Remove Tdistance2 from
                                      ;the symbol table

The #define directive was used to define two substitution strings: Tdistance1 to substitute for 50 and Tdistance2 to substitute for 25. Then #undefine was used to remove Tdistance2 from the symbol table, i.e., Tdistance2 can no longer be used to substitute for 25.

udata 0x20                               ;Set up distance_reg
distance_reg res 1                      ;at GPR 0x20

The udata and res directives are used to assign distance_reg to register 0x20. For more on these directives, see example 1.

rst code  0x00                          ;Reset Vector
   pagesel Start
   goto    Start
   code
   ;Start Program
   Start
   clrf    distance_reg                  ;Clear register
   movlw   Tdistance1                    ;Move value of Tdistance1
movwf distance_reg ;into distance_reg
constant distance1=10 ;Declare distance1
;a constant symbol

Declare a constant symbol, distance1, with a value of 10. Once a constant is declared, its value cannot be altered.

variable distance2 ;Declare distance2
;a variable symbol

Declare a variable symbol, distance2. The variable directive does not require the symbol to be initialized when declared.

distance3 set 10 ;Define a value for
;the symbol distance3

Define symbol distance3 as 10.

distance2=15 ;Give distance2 an
;initial value
distance2=distance1+distance2 ;Add distance1
;to distance2

Variable assignments, increments and decrements must be placed on separate lines.

distance3 set 15 ;Change value of distance3
distance2=distance1+distance2 ;Add distance1
;to distance2

movlw distance2 ;Move value of distance2
movwf distance_reg ;into distance_reg

goto Start ;Loop back to Start
end

5.4.3 Using Watch Windows

Once the program begins, the value of Tdistance1 is placed into distance_reg. This can be observed in a watch window in MPLAB IDE, where the value of distance_reg will become 50. The symbol Tdistance1 will not be found in the watch window symbol list, as symbols defined using the #define directive are not available for viewing in MPLAB IDE because they are not RAM variables.

The final lines of the example program write the final value of distance2 to distance_reg. If you had a watch window open to see distance_reg loaded with the value of 50, you will see it change to 3A. Remember that the radix is hexadecimal, so hex addition was used to determine the distance2 value.
5.5 EXAMPLE OF A 32-BIT DELAY ROUTINE

Directives highlighted in this example are:

• macro, endm
• banksel

5.5.1 Program Functional Description

A delay routine is needed in many applications. For this example, delay increments are 20 us, with the routine having a range of 40 us to 23.8 hours. (This assumes a 4 MHz clock.)

5.5.2 Commented Code Listing

;Each loop takes 20 clocks, or 20 us per loop, ;at 4MHz or 1MIPS clock.
;Turn off in config bits WDT for long simulations

#include p16F877A.inc

udata 0x20

Dly0  res   1    ;Stores 4 bytes of data for the delay count
Dly1  res   1    ;Dly0 is the least significant byte
Dly2  res   1    ;while Dly3 is the most significant byte
Dly3  res   1

Dly32 MACRO DLY
  goto $+1  ;delay 2 cycles
  goto $+1  ;delay total of 4 cycles

;Take the delay value argument from the macro, precalculate ;the required 4 RAM values and load the the RAM values Dly3 ;though Dly0.
BANKSEL Dly3
  movlw   (DLY-1) & H'FF'
  movwf   Dly0
  movlw   (DLY-1) >>D'08' & H'FF'
  movwf   Dly1
  movlw   (DLY-1) >>D'16' & H'FF'

;Bytes are shifted and anded by the assembler to make user ;calculations easier.
  movwf   Dly2
  movlw   (DLY-1) >>D'24' & H'FF'

;Call DoDly32 to run the delay loop.
  movwf   Dly3
  call    DoDly32
ENDM        ;End of Macro definition

RST CODE 0x00 ;Reset Vector
  pagesel TestCode
  goto    TestCode

CODE        ;Code starts here
  TestCode
  Dly32  D'50000' ;Max 4 billion+ (runs Dly32 Macro, ;1 sec in this case).
  nop      ;ZERO STOPWATCH, put breakpoint here.
goto TestCode ;Go back to top of program and ;run the delay again.

;Subroutine, called by the Macro Dly32 (20 Tcy per loop)
DoDly32

    movlw H'FF' ;Start with -1 in W
    addwf Dly0,F ;LSB decrement
    btfsc STATUS,C ;was the carry flag set?
    clrw ;If so, 0 is put in W
    addwf Dly1,F ;Else, we continue.
    btfsc STATUS,C
    clrw ;0 in W
    addwf Dly2,F
    btfsc STATUS,C
    clrw ;0 in W
    addwf Dly3,F
    btfsc STATUS,C
    clrw ;0 in W
    iorwf Dly0,W ;Inclusive-OR all variables
    iorwf Dly1,W ;together to see if we have reached
    iorwf Dly2,W ;0 on all of them.
    iorwf Dly3,W

    btfss STATUS,Z ;Test if result of Inclusive-OR's is 0
    goto DoDly32 ;It was NOT zero, so continue counting
    retlw 0 ;It WAS zero, so exit this subroutine.

END
5.6 EXAMPLE OF SPI EMULATED IN FIRMWARE

Directives highlighted in this example are:

- list
- #define
- udata, res
- global

5.6.1 Program Functional Description

This program is used to emulate SPI function in firmware.

The application is written as relocatable code, i.e., you must use both the assembler and linker to generate the executable.

The list directive is used to define the processor and set listing file formatting. The standard header file for the processor selected is included using include. SPI variables are declared using #define. Program registers are assigned using the udata and res directives. Sections of code are created using the code statement. External code is accessed using global.

5.6.2 Commented Code Listing

;********************************************************************
; Emulates SPI in firmware
; Place byte in Buffer, call SPI_Out - sends MSB first
;********************************************************************
LIST        P=18F4520       ;define processor
#include    <P18F4520.INC>  ;include file
list        c=132, n=0      ;132 col, no paging

;********************************************************************
#define Clk LATB,0  ; SPI clock output
#define Dat LATB,1  ; SPI data output
#define Bus LATB,2  ; busy indicator

;********************************************************************
;Variable definitions
udata
Buffer    res 1    ; SPI transmit data
Counter   res 1    ; SPI transmit bit counter
DelayCtr  res 1

;********************************************************************
code
SPI_Out
clrf    Counter                 ; init bit counter
bsf     Counter,7
bcf     Clk                     ; clear clock
bcf     Dat                     ; clear data out
bsf     Bus                     ; indicate busy
Lup     movf    Counter,W               ; get mask
         andwf   Buffer,W                ; test selected bit
         btfss   STATUS,Z                ; was result zero?
         bsf     Dat                     ; set data
         bsf     Clk                     ; set clock
         bcf     Clk                     ; clear clock
         bcf     Dat                     ; clear data
         rrncf   Counter,F               ; test next bit
         btfss   Counter,7               ; done with byte?
         bra     Lup                     ; no
         bcf     Bus                     ; indicate not busy

         return

;********************************************************************

         global  SPI_Out, Buffer

         end
5.7 EXAMPLE OF HEXADECIMAL TO ASCII CONVERSION

Directives highlighted in this example are:
- udata, res
- global

5.7.1 Program Functional Description

This program converts a hexadecimal byte into two ASCII bytes.

The application is written as relocatable code, i.e., you must use both the assembler and linker to generate the executable.

Program registers are assigned using the udata and res directives. Sections of code are created using the code statement. External code is accessed using global.

5.7.2 Commented Code Listing

```assembly
 Variables udata
 HexTemp res 1
 ASCIIH res 1
 ASCIIL res 1

 code
 Hex2ASC
 movf HexTemp,W
 andlw 0x0F
 call DecHex
 movwf ASCIIH

 swapf HexTemp,F
 movf HexTemp,W
 andlw 0x0F
 call DecHex
 movwf ASCIIL

 return

 DecHex
 sublw 0x09
 btfss STATUS,C
 goto HexC

 Dec
 movf HexTemp,W
 andlw 0x0F
 addlw A'0'
 return

 HexC
 movf HexTemp,W
 andlw 0x0F
 addlw A'A'-0x0A
 return
```
5.8 OTHER SOURCES OF EXAMPLES

Short examples of use for each directive are listed under each directive topic. See Chapter 4. “Directives”.

Examples of use for multiple directives are available from the following sources:

- readme.asm - Serial EEPROM Support
- Application Notes, Technical Briefs
  - Website - http://www.microchip.com
- Code Examples and Templates
  - MPLAB IDE installation directory
  - Website - http://www.microchip.com

5.9 TIPS AND TRICKS

To reduce costs, designers need to make the most of the available program memory in MCUs. Program memory is typically a large portion of the MCU cost. Optimizing the code helps to avoid buying more memory than needed. Here are some ideas that can help reduce code size. For more information, see Tips ’n Tricks (DS40040).

- TIP #1: Delay Techniques
- TIP #2: Optimizing Destinations
- TIP #3: Conditional Bit Set/Clear
- TIP #4: Swap File Register with W
- TIP #5: Bit Shifting Using Carry Bit
- TIP #6: Using External Memory

5.9.1 TIP #1: Delay Techniques

- Use GOTO Next Instruction instead of two NOPs.
- Use CALL Rtrn as quad, 1 instruction NOP (where Rtrn is the exit label from existing subroutine).

```assembly
;******************************************************************************
global Hex2ASC, ASCIIH, ASCIIL

END

NOP              ;2 instructions, 2 cycles
GOTO $+1         ;1 instruction, 2 cycles
Call Rtrn       ;1 instruction, 4 cycles
: Rtrn RETURN
;******************************************************************************
```

MCUs are commonly used to interface with the “outside world” by means of a data bus, LED’s, buttons, latches, etc. Because the MCU runs at a fixed frequency, it will often need delay routines to meet setup/hold times of other devices, pause for a handshake or decrease the data rate for a shared bus.
Longer delays are well-suited for the DECFSZ and INCFSZ instructions where a variable is decremented or incremented until it reaches zero when a conditional jump is executed. For shorter delays of a few cycles, here a few ideas to decrease code size.

For a two cycle delay, it is common to use two NOP instructions which uses two program memory locations. The same result can be achieved by using GOTO $+1. The $ represents the current program counter value in MPASM assembler. When this instruction is encountered, the MCU will jump to the next memory location. This is what it would have done if two NOP’s were used, but since the GOTO instruction uses two instruction cycles to execute, a two-cycle delay was created. This created a two-cycle delay using only one location of program memory.

To create a four cycle delay, add a label to an existing RETURN instruction in the code. In this example, the label Rtrn was added to the RETURN of subroutine that already existed somewhere in the code. When executing CALL Rtrn, the MCU delays two instruction cycles to execute the CALL and two more to execute the RETURN. Instead of using four NOP instructions to create a four cycle delay, the same result was achieved by adding a single CALL instruction.

5.9.2 TIP #2: Optimizing Destinations

- Destination bit determines \( W \) or \( F \) for result
- Look at data movement and restructure

Example: \( A + B \rightarrow A \)

```
MOVF A,WMOVF B,W
ADDWF B,W
ADDWF A,F
MOVWF A
```

3 instructions2 instructions

Careful use of the destination bits in instructions can save program memory. Here, register A and register B are summed and the result is put into the A register. A destination option is available for logic and arithmetic operations. In the first example, the result of the ADDWF instruction is placed in the working register. A MOVWF instruction is used to move the result from the working register to register A. In the second example, the ADDWF instruction uses the destination bit to place the result into the A register saving an instruction.

5.9.3 TIP #3: Conditional Bit Set/Clear

- To move single bit of data from REGA to REGB
- Precondition REGB bit
- Test REGA bit and fix REGB if necessary

```
BTFSS REGA,2BCF   REGB,5
BCF   REGB,5BTFSC REGA,2
BTFSC REGA,2BSF   REGB,5
BSF   REGB,5
```

4 instructions3 instructions

One technique for moving one bit from the REGA register to REGB is to perform bit tests. In the first example, the bit in REGA is tested using a BTFSS instruction. If the bit is clear, the BCF instruction is executed and clears the REGB bit, and if the bit is set, the instruction is skipped. The second bit test determines if the bit is set, and if so, will execute the BSF and set the REGB bit, otherwise the instruction is skipped. This sequence requires four instructions.
A more efficient technique is to assume the bit in REGA is clear, and clear the REGB bit, and test if the REGA bit is clear. If so, the assumption was correct and the BSF instruction is skipped, otherwise the REGB bit is set. The sequence in the second example uses three instructions because one bit test was not needed.

One important point is that the second example will create a two cycle glitch if REGB is a port outputting a high. This is caused by the BCF and BTFSC instructions that will be executed regardless of the bit value in REGA.

5.9.4 TIP #4: Swap File Register with W

The following macro swaps the contents of W and REG without using a second register.

```
SWAPWF MACRO REG
    XORWF REG,F
    XORWF REG,W
    XORWF REG,F
ENDM
```

Needs: 0 TEMP registers, 3 Instructions, 3 Tcy

An efficient way of swapping the contents of a register with the working register is to use three XORWF instructions. It requires no temporary registers and three instructions. Here’s an example:

<table>
<thead>
<tr>
<th>W</th>
<th>REG</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101100</td>
<td>01011100</td>
<td>XORWF REG,F</td>
</tr>
<tr>
<td>10101100</td>
<td>11110000</td>
<td>XORWF REG,W</td>
</tr>
<tr>
<td>01011100</td>
<td>11110000</td>
<td>XORWF REG,F</td>
</tr>
<tr>
<td>01011100</td>
<td>10101100</td>
<td>Result</td>
</tr>
</tbody>
</table>

5.9.5 TIP #5: Bit Shifting Using Carry Bit

Rotate a byte through carry without using RAM variable for loop count:

- Easily adapted to serial interface transmit routines.
- Carry bit is cleared (except last cycle) and the cycle repeats until the zero bit sets indicating the end.

```
list p=12f629
#include p12f629.inc
buffer equ 0x20
bsf STATUS,C    ;Set ‘end of loop’ flag
rlf buffer,F    ;Place first bit into C
Send_Loop
    bcf GPIO,Dout ;Precondition output
    btfsc STATUS,C ;Check data - 0 or 1?
    bsf GPIO,Dout
    bcf STATUS,C ;Clear data in C
    rlf buffer,F ;Place next bit into C
    movf buffer,F ;Force Z bit
    btfss STATUS,Z ;Exit?
    goto Send_Loop
```

Related Topic: TIP #3: Conditional Bit Set/Clear
5.9.6 TIP #6: Using External Memory

To use external memory, the maximum allowable address must be redefined by using the _MAXROM directive. For example, when using the PIC18F87J10 in extended microcontroller mode, the _MAXROM directive must be used as follows:

```
#include <P18cxxx.inc>

_MAXROM 0x1FFFFF

; 87J10 Configuration for external memory
   CONFIG MODE=XM20, EASHFT=OFF, BW = 16, WAIT=OFF

org 0x0000
goto 0x10000

END
```
Chapter 6. Relocatable Objects

6.1 INTRODUCTION

MPASM assembler, used with MPLINK object linker, has the ability to generate and link precompiled object modules. Writing source code that will be assembled to an object module is slightly different from writing code used to generate an executable (hex) file directly. MPASM assembler routines designed for absolute address assembly will require minor modifications to compile correctly into relocatable object modules.

Topics covered in this chapter:
• Header Files
• Program Memory
• Low, High and Upper Operators
• RAM Allocation
• Configuration Bits and ID Locations
• Accessing Labels From Other Modules
• Paging and Banking Issues
• Generating the Object Module
• Code Example

6.2 HEADER FILES

The Microchip-supplied standard header files (e.g., p18f8720.inc) should be used when generating object modules. These header files define the special function registers for the target processor.

EXAMPLE 6-1: INCLUDE HEADER FILE

```
#include p18f8720.inc
```

See 4.42 "#include - Include Additional Source File" for more information.
6.3 PROGRAM MEMORY

Program memory code must be organized into a logical code section. To do this, the code must be preceded by a code section declaration (See 4.9 "code - Begin an Object File Code Section") to make it relocatable.

<table>
<thead>
<tr>
<th>Absolute Code</th>
<th>Equivalent Relocatable Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start clrw</td>
<td>code ;Address determined ;by the linker.</td>
</tr>
<tr>
<td>option</td>
<td>Start clrw</td>
</tr>
<tr>
<td></td>
<td>option</td>
</tr>
<tr>
<td>Progl org 0x0100 movlw 0x0A movwf var1</td>
<td>Progl code 0x0100 ;Start at 0x0100 movlw 0x0A movwf var1</td>
</tr>
</tbody>
</table>

If more than one code section is defined in a source file, each section must have a unique name. If the name is not specified, it will be given the default name .code.

Each program memory section must be contiguous within a single source file. A section may not be broken into pieces within a single source file.

The physical address of the code can be fixed by supplying the optional address parameter of the code directive. Situations where this might be necessary are:

- Specifying reset and interrupt vectors
- Ensuring that a code segment does not overlap page boundaries

EXAMPLE 6-2: RELOCATABLE CODE

Reset code 0x01FF ;Fixed address
goto Start
Pgm code ;Address determined by the linker
clrw
option

6.4 LOW, HIGH AND UPPER OPERATORS

Low, high and upper operators are used to return one byte of a multi-byte label value. If low is used, only bits 0 through 7 of the expression will be used. If high is used, only bits 8 through 15 of the expression will be used. If upper is used, only bits 16 through 21 of the expression will be used.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>Return low byte of operand.</td>
</tr>
<tr>
<td>high</td>
<td>Return high byte of operand.</td>
</tr>
<tr>
<td>upper</td>
<td>Return upper byte of operand.</td>
</tr>
<tr>
<td>scnsz_low</td>
<td>Return low byte of section size.</td>
</tr>
<tr>
<td>scnsz_high</td>
<td>Return high byte of section size.</td>
</tr>
<tr>
<td>scnsz_upper</td>
<td>Return upper byte of section size.</td>
</tr>
<tr>
<td>scnend_low</td>
<td>Return low byte of section end operand.</td>
</tr>
<tr>
<td>scnend_high</td>
<td>Return high byte of section end operand.</td>
</tr>
<tr>
<td>scnend_upper</td>
<td>Return upper byte of section end operand.</td>
</tr>
<tr>
<td>scnsstart_low</td>
<td>Return low byte of section start operand.</td>
</tr>
<tr>
<td>scnsstart_high</td>
<td>Return high byte of section start operand.</td>
</tr>
<tr>
<td>scnsstart_upper</td>
<td>Return upper byte of section start operand.</td>
</tr>
</tbody>
</table>

Operator precedence information may be found in 3.5 "Arithmetic Operators and Precedence".
There are some restrictions involving these operators with relocatable symbols. For example, the low, high and upper operators must be of the form:

\[ \text{[low|high|upper] (relocatable\_symbol + constant\_offset)} \]

where:

- \textit{relocatable\_symbol} is any label that defines a program or data memory address
- \textit{constant\_offset} is an expression that is resolvable at assembly time to a value between -32768 and 32767

Either \textit{relocatable\_symbol} or \textit{constant\_offset} may be omitted.

Operands of the form:

\[ \text{relocatable\_symbol - relocatable\_symbol} \]

will be reduced to a constant value if both symbols are defined in the same code or data section.

In addition to section operators, there are section pseudo-instructions.

<table>
<thead>
<tr>
<th>Pseudo-Instruction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>scnend_lfsr</td>
<td>scnend_lfsr n,s, where n is 0, 1, or 2 (as with the LFSR instruction) and s is a string which is taken to be the name of a section. This instruction loads LFSR with the end address of the section.</td>
</tr>
<tr>
<td>scnstart_lfsr</td>
<td>scnstart_lfsr n,s, where n is 0, 1, or 2 (as with the LFSR instruction) and s is a string which is taken to be the name of a section. This instruction loads LFSR with the start address of the section.</td>
</tr>
</tbody>
</table>

These operators and instructions only have meaning when an object file is generated; they cannot be used when generating absolute code.

**EXAMPLE 6-3: GENERAL OPERATOR USE**

The general operators, low, high and upper, may be used to access data in tables. The following code example was taken the p18demo.asm file provided with PICDEM 2 Plus demo board. The excerpt shows how “Microchip” is read from the table and displayed on the demo board LCD.

```assembly
#include p18f452.inc

PROG1 CODE

stan_table ;table for standard code
            "XXXXXXXXXXXXXXXXX"
            ptr:
data " Volmeter " ;0
data " Buzzer " ;16
data " Temperature " ;32
data " Clock " ;48
data "RA4=Next RB0=Now" ;64
data " Microchip " ;80
data " PICDEM 2 PLUS " ;96
data "RA4=Set RB0=Menu" ;112
data "RA4= --> RB0= ++" ;128
data " RB0 = Exit " ;144
data "Volts = " ;160
data "Prd.=128 DC=128 " ;176
```
;********************************************************** STANDARD CODE MENU SELECTION **********************************************************

    movlw   .80                 ;send "Microchip" to LCD
    movwf   ptr_pos
    call    stan_char_1
    :
    ;----Standard code, Place characters on line-1----

stan_char_1
    call    LCDLine_1           ;move cursor to line 1
    movlw   .16                 ;1-full line of LCD
    movwf   ptr_count
    movlw   UPPER stan_table    ;use operators to load
    movwf   TBLPTRU             ;table pointer values
    movlw   HIGH stan_table
    movwf   TBLPTRH
    movlw   LOW stan_table
    movwf   TBLPTRL
    movf    ptr_pos,W
    addwf   TBLPTRL,F
    clrf    WREG
    addwfc  TBLPTRH,F
    addwfc  TBLPTRU,F

stan_next_char_1
tblrd   *+
    movff   TABLAT,temp_wr
    call    d_write              ;send character to LCD
    decfsz  ptr_count,F         ;move pointer to next char
    bra      stan_next_char_1

movlw   "\n"                 ;move data into TXREG
movwf   TXREG              ;next line
btfss   TXSTA,TRMT         ;wait for data TX
    goto    $-2
movlw   "\r"                 ;move data into TXREG
movwf   TXREG              ;carriage return
btfss   TXSTA,TRMT         ;wait for data TX
    goto    $-2

    return
    :

6.5 RAM ALLOCATION

RAM space must be allocated in a data section. Five types of data sections are available:

**Note:** The ability to use access, overlaid or shared data varies by device. Consult your device data sheet for more information.

- **udata** - Uninitialized data. This is the most common type of data section. Locations reserved in this section are not initialized and can be accessed only by the labels defined in this section or by indirect accesses. See 4.62 "udata - Begin an Object File Uninitialized Data Section".
- **udata_acs** - Uninitialized access data. This data section is used for variables that will be placed in access RAM of PIC18 devices. Access RAM is used as quick data access for specified instructions. See 4.63 "udata_acs - Begin an Object File Access Uninitialized Data Section (PIC18 MCUs)".
- **udata_ovr** - Uninitialized overlaid data. This data section is used for variables that can be declared at the same address as other variables in the same module or in other linked modules. A typical use of this section is for temporary variables. See 4.64 "udata_ovr - Begin an Object File Overlaid Uninitialized Data Section".
- **udata_shr** - Uninitialized shared data. This data section is used for variables that will be placed in RAM of PIC12/16 devices that is unbanked or shared across all banks. See 4.65 "udata_shr - Begin an Object File Shared Uninitialized Data Section (PIC12/16 MCUs)".
- **idata** - Initialized data. The linker will generate a lookup table that can be used to initialize the variables in this section to the specified values. When linked with MPLAB C17 or C18 code, these locations will be initialized during execution of the startup code. The locations reserved by this section can be accessed only by the labels defined in this section or by indirect accesses. See 4.36 "idata - Begin an Object File Initialized Data Section".

The following example shows how a data declaration might be created.

**EXAMPLE 6-4: RAM ALLOCATION**

**Absolute Code**

Use `cblock` to define variable register locations (See 4.8 "cblock - Define a Block of Constants"). Variable values will need to be specified in code.

```assembly
cblock 0x20
    HistoryVector   ;Must be initialized to 0
    InputGain, OutputGain ;Control loop gains
    Templ, Temp2, Temp3 ;Used for internal calculations
endc
```

**Equivalent Relocatable Code**

Use data declarations to define register locations and initialize.

```assembly
idata
    HistoryVector db 0       ;Initialized to 0
udata
    InputGain  res 1         ;Control loop gains
    OutputGain res 1
udata_ovr
    Templ res 1             ;Used for internal calculations
    Temp2 res 1
    Temp3 res 1
```

Note: The ability to use access, overlaid or shared data varies by device. Consult your device data sheet for more information.
If necessary, the location of the section may be fixed in memory by supplying the optional address parameter. If more than one of each section type is specified, each section must have a unique name. If a name is not provided, the default section names are: .idata, .udata, .udata_acs, .udata_shr, and .udata_ovr.

When defining initialized data in an .idata section, the directives db, dw, and data can be used. db will define successive bytes of data memory. dw and data will define successive words of data memory in low-byte/high-byte order. The following example shows how data will be initialized.

```
EXAMPLE 6-5: RELOCATABLE CODE LISTING

00001 IDATA
0000 01 02 03    00002 Bytes DB 1,2,3
0003 34 12 78 56 00003 Words DW 0x1234,0x5678
0007 41 42 43 00 00004 String DB "ABC", 0
```

6.6 CONFIGURATION BITS AND ID LOCATIONS

Configuration bits and ID locations can still be defined in a relocatable object using the following directives:

- Section 4.11 “__config - Set Processor Configuration Bits”
- Section 4.12 “config - Set Processor Configuration Bits (PIC18 MCUs)”
- Section 4.38 “__idlocs - Set Processor ID Locations”

Only one linked module can specify these directives. They should be used prior to declaring any code sections. After using these directives, the current section is undefined.

6.7 ACCESSING LABELS FROM OTHER MODULES

Labels that are defined in one module for use in other modules must be exported using the global directive (see 4.35 “global - Export a Label”). Modules that use these labels must use the extern directive (see 4.33 “extern - Declare an Externally Defined Label”) to declare the existence of these labels. An example of using the global and extern directives is shown below.

```
EXAMPLE 6-6: RELOCATABLE CODE, DEFINING MODULE

udata
  InputGain  res 1
  OutputGain res 1
global InputGain, OutputGain
code
  Filter
  global Filter
    ; Filter code
```

```
EXAMPLE 6-7: RELOCATABLE CODE, REFERENCING MODULE

extern InputGain, OutputGain, Filter
code
  Reading res 1
```
```
    ;
    movlw GAIN1
    movwf InputGain
    movlw GAIN2
    movwf OutputGain
```
6.8 PAGING AND BANKING ISSUES

In many cases, RAM allocation will span multiple banks, and executable code will span multiple pages. In these cases, it is necessary to perform proper bank and page set-up to properly access the labels. However, since the absolute addresses of these variable and address labels may not be known at assembly time, it is not always possible to place the proper code in the source file. For these situations two directives, banksel (4.7 “banksel - Generate Bank Selecting Code”) and pagesel (4.53 “pagesel - Generate Page Selecting Code (PIC10/12/16 MCUs)”), have been added. These directives instruct the linker to generate the correct bank or page selecting code for a specified label. An example of how code should be converted is shown below.

EXAMPLE 6-8: BANKSEL AND PAGESEL

Hard-Coded Banking and Paging

Use indirect addressing (FSR) and the Status register for banking and paging, respectively.

```assembly
#include p12f509.inc
Var1 equ 0x10 ;Declare variables
Var2 equ 0x30
...
movlw InitialValue
bcf FSR, 5 ;Data memory Var1 bank (0)
movwf Var1
movf Varl
bsf FSR, 5 ;Data memory Var2 bank (1)
movwf Var2
bsf STATUS, PA0 ;Program memory page 1
call Subroutine
...
Subroutine clrw ;On Page 1
...
retlw 0
```

BANKSEL for Banking and PAGESEL for Paging

Use banksel and pagesel for banking and paging, respectively.

```assembly
#include p12f509.inc
extern Var1, Var2 ;Declare variables

code
movlw InitialValue
banksel Var1 ;Select data memory Var1 bank
movwf Var1
banksel Var2 ;Select data memory Var2 bank
movwf Var2
pagesel Subroutine ;Select program memory page
call Subroutine
...
Subroutine clrw ;Page unknown at assembly time
...
retlw 0
```
6.9 GENERATING THE OBJECT MODULE

Once the code conversion is complete, the object module is generated automatically in MPLAB IDE or by requesting an object file on the command line or in the shell interface. When using MPASM assembler for Windows, check the checkbox labeled "Object File". When using the command line interface, specify the /o option. The output file will have a .o extension.

6.10 CODE EXAMPLE

Since an eight-by-eight bit multiply is a useful, generic routine, it would be handy to break this off into a separate object file that can be linked in when required. The absolute code file can be broken into two relocatable code files: a calling file representing an application and a generic routine that could be incorporated in a library.

This code was adapted from application note AN617. Please see the Microchip website for a downloadable PDF of this application note.

EXAMPLE 6-9: ABSOLUTE CODE

; Input: fixed point arguments in AARGB0 and BARGB0
; Output: product AARGB0xBARGB0 in AARGB0:AARGB1
; Other comments truncated. See AN617.
;********************************************************************
#include   p16f877a.inc ;Use any PIC16 device you like
LOOPCOUNT  EQU   0x20  ;7 loops needed to complete routine
AARGB0     EQU   0x21  ;MSB of result out,
AARGB1     EQU   0x22  ;operand A in (8 bits)
BARGB0     EQU   0x23  ;LSB of result out,
;operand B in (8 bits)
TestCode
    clrf    AARGB1      ;Clear partial product before testing
    movlw   D'11'
    movwf   AARGB0
    movlw   D'30'
    movwf   BARGB0
    call    UMUL0808L   ;After loading AARGB0 and BARGB0,
;call routine
    goto    $           ;Result now in AARGB0:AARGB1,
;where (B0 is MSB)
END

UMUL0808L
    movlw   0x08
    movwf   LOOPCOUNT
    movf    AARGB0,W
_LOOPUM0808A
    rrf     BARGB0, F
    btfsc   STATUS,C
    goto    LUM0808NAP
    decfsz  LOOPCOUNT, F
    goto    _LOOPUM0808A
    clrf    AARGB0
    retlw   0x00
LUM0808NAP
    bcf     STATUS,C
    goto    LUM0808NA

EXAMPLE 6-10: RELOCATABLE CODE, CALLING FILE

; Input: fixed point arguments in AARGB0 and BARGB0
; Output: product AARGxBARG in AARGB0:AARGB1
; Other comments truncated. See AN617.

#include p16f877a.inc ;Use any PIC16 device you like
EXTERN UMUL0808L, AARGB0, AARGB1, BARGB0

Reset CODE 0x0
pagesel TestCode
goto TestCode

CODE
TestCode
banksel AARGB1
clrfl AARGB1 ;Clear partial product before testing
movlw D'11' ;Load in 2 test values
movwf AARGB0
movlw D'30'
movwf BARGB0
pagesel UMUL0808L
 call UMUL0808L ;After loading AARGB0 and BARGB0, ;call routine
goto $ ;Result now in AARGB0:AARGB1, ;where (AARGB0 is MSB)

END

EXAMPLE 6-11: RELOCATABLE CODE, LIBRARY ROUTINE

; Input: fixed point arguments in AARGB0 and BARGB0
; Output: product AARGxBARG in AARGB0:AARGB1
; Other comments truncated. See AN617.

#include p16f877a.inc ;Use any PIC16 device you like
GLOBAL UMUL0808L, AARGB0, AARGB1, BARGB0

UDATA
LOOPCOUNT RES 1 ;7 loops needed to complete routine
AARGB0 RES 1 ;MSB of result out,
AARGB1 RES 1 ;operand A in (8 bits)
BARGB0 RES 1 ;LSB of result out,
BARGB1 RES 1 ;operand B in (8 bits)
CODE
UMUL0808L
    movlw 0x08
    movwf LOOPCOUNT
    movf AARGBO,W
LOOPUM0808A
    rrf BARGBO, F
    btfsc STATUS,C
    goto LUM0808NAP
    decfsz LOOPCOUNT, F
    goto LOOPUM0808A
    clrf AARGBO
    retlw 0x00
LUM0808NAP
    bcf STATUS,C
    goto LUM0808NA
LOOPUM0808
    rrf BARGBO, F
    btfsc STATUS,C
    addwf AARGBO, F
LUM0808NA
    rrf AARGBO, F
    rrf AARGBO1, F
    decfsz LOOPCOUNT, F
    goto LOOPUM0808
    retlw 0

END
Chapter 7. Macro Language

7.1 INTRODUCTION

Macros are user defined sets of instructions and directives that will be evaluated in-line with the assembler source code whenever the macro is invoked. Macros consist of sequences of assembler instructions and directives. They can be written to accept arguments, making them quite flexible. Their advantages are:

- Higher levels of abstraction, improving readability and reliability.
- Consistent solutions to frequently performed functions.
- Simplified changes.
- Improved testability.

Applications might include creating complex tables, frequently used code, and complex operations.

Topics covered in this chapter:
- Macro Syntax
- Macro Directives Defined
- Macro Definition
- Macro Invocation
- Macro Code Examples

7.2 MACRO SYNTAX

MPASM assembler macros are defined according to the following syntax:

```
label macro [arg1, arg2 ..., argn]
    : ...
endm
```

where `label` is a valid assembler label that will be the macro name and `arg` is any number of optional arguments supplied to the macro (that will fit on the source line.) The values assigned to these arguments at the time the macro is invoked will be substituted wherever the argument name occurs in the body of the macro.

The body of a macro may be comprised of MPASM assembler directives, PIC1X MCU assembly instructions, or MPASM assembler macro directives (local for example.) The assembler continues to process the body of the macro until an `exitm` or `endm` directive is encountered.

**Note:** Macros must be defined before they are used, i.e., forward references to macros are not permitted.
7.3 MACRO DIRECTIVES DEFINED

There are directives that are unique to macro definitions. They cannot be used out of the macro context.

- **4.45 "macro - Declare Macro Definition"
- **4.31 "exitm - Exit from a Macro"
- **4.26 "endm - End a Macro Definition"
- **4.32 "expand - Expand Macro Listing"
- **4.49 "noexpand - Turn off Macro Expansion"
- **4.44 "local - Declare Local Macro Variable"

When writing macros, you can use any of these directives PLUS any other directives supported by the assembler.

Note: The previous syntax of the "dot" format for macro specific directives is no longer supported.

7.4 MACRO DEFINITION

String replacement and expression evaluation may appear within the body of a macro.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arg</code></td>
<td>Substitute the argument text supplied as part of the macro invocation.</td>
</tr>
<tr>
<td><code>#v(expr)</code></td>
<td>Return the integer value of <code>expr</code>. Typically used to create unique variable names with common prefixes or suffixes. Cannot be used in conditional assembly directives (e.g. <code>ifdef</code>, <code>while</code>).</td>
</tr>
</tbody>
</table>

Arguments may be used anywhere within the body of the macro, except as part of normal expression.

The `exitm` directive provides an alternate method for terminating a macro expansion. During a macro expansion, this directive causes expansion of the current macro to stop and all code between the `exitm` and the `endm` directives for this macro to be ignored. If macros are nested, `exitm` causes code generation to return to the previous level of macro expansion.

7.5 MACRO INVOCATION

Once the macro has been defined, it can be invoked at any point within the source module by using a macro call, as described below:

```
macro_name [arg, ..., arg]
```

where `macro_name` is the name of a previously defined macro and arguments are supplied as required.

The macro call itself will not occupy any locations in memory. However, the macro expansion will begin at the current memory location. Commas may be used to reserve an argument position. In this case, the argument will be an empty string. The argument list is terminated by white space or a semicolon.
EXAMPLE 7-1: MACRO CODE GENERATION

The following macro:

```assembly
define_table macro
    local a = 0
    while a < 3
        entry#v(a) dw 0
        a += 1
    endw
endm
```

When invoked, would generate:

```assembly
text0 dw 0
text1 dw 0
text2 dw 0
```

7.6 MACRO CODE EXAMPLES

The following are examples of macros:

- Literal to RAM Conversion
- Constant Compare

7.6.1 Literal to RAM Conversion

This code converts any literal of 32 bits to 4 separate RAM data values. In this example, the literal 0x12345678 is put in the desired 8 bit registers as 0x12, 0x34, 0x56, and 0x78. Any literal can be “unpacked” this way using this macro.

```assembly
#include   p16F877A.inc

udata 0x20
Out0   res   1   ; LSB
Out1   res   1   ; :
Out2   res   1   ; :
Out3   res   1   ; MSB

Unpack32 MACRO Var, Address ;Var = 32 bit literal to be unpacked
    BANKSEL Address            ;Address specifies the LSB start
    movlw   Address            ;Use FSR and INDF for indirect
    movwf   FSR                ;access to desired address

    movlw   Var & H'FF'        ;Mask to get LSB
    movwf   INDF               ;Put in first location

    movlw   Var >>D'08' & H'FF';Mask to get next byte of literal
    incf    FSR,F              ;Point to next byte

    movwf   INDF               ;Write data to next byte

    movlw   Var >>D'16' & H'FF';Mask to get next byte of literal
    incf    FSR,F              ;Point to next byte

    movwf   INDF               ;Write data to next byte

    movlw   Var >>D'24' & H'FF';Mask to get last byte of literal
    incf    FSR,F              ;Point to last byte

    movwf   INDF               ;Write data to last byte
    ENDM                      ;End of the Macro Definition

ORG     0
Start                        ;TEST CODE for Unpack32 MACRO
    Unpack32 0x12345678,Out0  ;Put Unpack Macro here
    goto   $                  ;Do nothing (loop forever)
END
```
7.6.2 Constant Compare

As another example, if the following macro were written:

```assembly
#include "pic16f877a.inc"
;
; compare file to constant and jump if file
; >= constant.
;
cfl_jge macro file, con, jump_to
    movlw con & 0xff
    subwf file, w
    btfsc status, carry
    goto jump_to
endm
```

and invoked by:

```assembly
cfl_jge switch_val, max_switch, switch_on
```

it would produce:

```assembly
movlw max_switch & 0xff
subwf switch_val, w
btfsc status, carry
goto switch_on
```
8.1 INTRODUCTION

Error messages, warning messages and general messages produced by the MPASM assembler are listed and detailed here. These messages always appear in the listing file directly above each line in which the error occurred. Limitations of the assembler tool are also listed.

The messages are stored in the error file (.err) if no MPASM assembler options are specified. If the /e- option is used (turns error file off), then the messages will appear on the screen. If the /q (quiet mode) option is used with the /e-, then the messages will not display on the screen or in an error file. The messages will still appear in the listing file.

Topics covered in this chapter:
• Assembler Errors
• Assembler Warnings
• Assembler Messages
• Assembler Limitations

8.2 ASSEMBLER ERRORS

MPASM assembler errors are listed numerically below:

101 ERROR:
User error, invoked with the error directive.

102 Out of memory
Not enough memory for macros, #define's or internal processing.

103 Symbol table full
No more memory available for the symbol table.

104 Temp file creation error
Could not create a temporary file. Check the available disk space.
105 Cannot open file
Could not open a file. If it is a source file, the file may not exist. If it is an output file, the old version may be write protected.
To check for write-protect, right-click on the file named by MPLAB IDE in Windows. Choose “Properties” and see if “read-only” is checked. If it is, it cannot be modified by MPLAB IDE and will generate this error message. This often happens when you save your project to a CD-R or similar write-once media as a backup, and then copy the data to your computer. Copying to a CD marks all files as read-only (they cannot be changed on a CD-R), and when you copy the files, the attributes move with them making them all read-only on your hard drive. A good way to prevent this is to archive all of the files in one file, such as a *.ZIP, and then restore them from CD. The archive will preserve the original file attributes.

106 String substitution too complex
A string substitution was attempted that was too complex. Check for nesting of #define's.

107 Illegal digit
An illegal digit in a number. Valid digits are 0-1 for binary, 0-7 for octal, 0-9 for decimal, and 0-9, a-f, and A-F for hexadecimal.

108 Illegal character
An illegal character in a label. Valid characters for labels are alphabetic (a..f, A..F), numeric (0-9), the underscore (_), and the question mark (?). Labels may not begin with a numeric.

109 Unmatched (
An open parenthesis did not have a matching close parenthesis. For example, DATA (1+2.

110 Unmatched )
An close parenthesis did not have a matching open parenthesis. For example, DATA 1+2).

111 Missing symbol
An equ or set directive did not have a symbol to which to assign the value.

112 Missing operator
An arithmetic operator was missing from an expression. For example, DATA 1 2.
113 Symbol not previously defined
A symbol was referenced that has not yet been defined. Check the spelling and location of the declaration of any symbols used in your code. Only addresses may be used as forward references. Constants and variables must be declared before they are used.
This sometimes happens when #include files are used in your project. Since the text from an include file is inserted at the location of the #include statement, and you may have labels used before that point, you can get this error. Also, the error may occur due to a typing error, spelling mistake or case change in your label. MyLabel is not the same as Mylabel unless case sensitivity is turned off (it is on by default). Additionally, goto MyLabel will never locate the code at Mylabel or Mylable. Check for these sorts of mistakes first. As a general rule, put your include files at the top of each file. If this seems to cluttered, you may include files within other include files.

114 Divide by zero
Division by zero encountered during an expression evaluation.

115 Duplicate label
A label was declared as a constant (e.g., with the equ or cblock directive) in more than one location.

116 Address label duplicated or different in second pass
The same label was used in two locations. Alternately, the label was used only once but evaluated to a different location on the second pass. This often happens when users try to write page-bit setting macros that generate different numbers of instructions based on the destination.

117 Address wrapped around 0
For PIC12/16 devices, the location counter can only advance to 0xFFFF. After that, it wraps back to 0. Error 117 is followed by error 118.

118 Overwriting previous address contents
Code was previously generated for this address.

119 Code too fragmented
The code is broken into too many pieces. This error is very rare, and will only occur in source code that references addresses above 32K (including configuration bits).

120 Call or jump not allowed at this address
A call or jump cannot be made to this address. For example, CALL destinations on the PIC16C5x family must be in the lower half of the page.

121 Illegal label
Labels are not allowed on certain directive lines. Simply put the label on its own line, above the directive. Also, high, low, page, and bank are not allowed as labels.

122 Illegal opcode
Token is not a valid opcode.
123 Illegal directive
Directive is not allowed for the selected processor; for example, the __idlocs directive on devices with ID locations.

124 Illegal argument
An illegal directive argument; for example, list foobar.

125 Illegal condition
A bad conditional assembly. For example, an unmatched endif.

126 Argument out of range
Opcode or directive argument out of the valid range; for example, TRIS 10.

127 Too many arguments
Too many arguments specified for a macro call.

128 Missing argument(s)
Not enough arguments for a macro call or an opcode.

129 Expected
Expected a certain type of argument. The expected list will be provided.

130 Processor type previously defined
A different family of processor is being selected.

131 Processor type is undefined
Code is being generated before the processor has been defined. Note that until the processor is defined, the opcode set is not known.

132 Unknown processor
The selected processor is not a valid processor.

133 Hex file format INHX32 required
An address above 32K was specified.

134 Illegal hex file format
An illegal hex file format was specified in the list directive.

135 Macro name missing
A macro was defined without a name.

136 Duplicate macro name
A macro name was duplicated.

137 Macros nested too deep
The maximum macro nesting level was exceeded.

138 Include files nested too deep
The maximum include file nesting level was exceeded.
139 Maximum of 100 lines inside WHILE-ENDW
A while-endw can contain at most 100 lines.

140 WHILE must terminate within 256 iterations
A while-endw loop must terminate within 256 iterations. This is to prevent infinite assembly.

141 WHILEs nested too deep
The maximum while-endw nesting level was exceeded.

142 IFs nested too deep
The maximum if nesting level was exceeded.

143 Illegal nesting
Macros, if’s and while’s must be completely nested; they cannot overlap. If you have an if within a while loop, the endif must come before the endw.

144 Unmatched ENDC
endc found without a cblock.

145 Unmatched ENDM
endm found without a macro definition.

146 Unmatched EXITM
exitm found without a macro definition.

147 Directive/operation only allowed when generating an object file
The instruction/operand shown only has meaning when a linkable object file is generated. It cannot be used when generating absolute code.

148 Expanded source line exceeded 200 characters
The maximum length of a source line, after #define and macro parameter substitution, is 200 characters. Note that #define substitution does not include comments, but macro parameter substitution does.

149 Directive only allowed when generating an object file
Certain directives, such as global and extern, only have meaning when a linkable object file is generated. They cannot be used when generating absolute code.

150 Labels must be defined in a code or data section when making an object file
When generating a linkable object file, all data and code address labels must be defined inside a data or code section. Symbols defined by the equ and set directives can be defined outside of a section.

151 Operand contains unresolvable labels or is too complex
When generating an object file, operands must be of the form [high|low][relocatable address label]+[offset]).
152 Executable code and data must be defined in an appropriate section
When generating a linkable object file, all executable code and data declarations must be placed within appropriate sections.

153 Page or Bank bits cannot be evaluated for the operand
The operand of a pagesel, banksel or bankisel directive must be a relocatable address label or a constant.

154 Each object file section must be contiguous
Object file sections, except udata_ovr sections, cannot be stopped and restarted within a single source file. To resolve this problem, either name each section with its own name or move the code and data declarations such that each section is contiguous. This error will also be generated if two sections of different types are given the same name.

155 All overlaid sections of the same name must have the same starting address
If multiple udata_ovr sections with the same name are declared, they must all have the same starting address.

156 Operand must be an address label
When generating object files, only address labels in code or data sections may be declared global. Variables declared by the set or equ directives may not be exported.

157 ORG at odd address
For PIC18 devices, you cannot place org at an odd address, only even. Consult your device data sheet.

158 Cannot use RES directive with odd number of bytes
For PIC18 devices, you cannot use res to specify an odd number of bytes, only even. Consult your device data sheet.

159 Cannot use FILL directive with odd number of bytes
For PIC18 devices, you cannot use fill to fill with data an odd number of bytes, only even. Consult your device data sheet.

160 CODE_PACK directive not available for this part; substituting CODE
The code_pack directive can only be used with byte-addressable ROM.

161 Non-negative value required for this context.
Some contexts require non-negative values.

162 Expected a section name
Some operators and pseudo-operators take section names as operands. The lexical form of a section name is that of an identifier, optionally prefixed with a ‘.’.

163 __CONFIG directives must be contiguous
Do not place other code between __config directive declarations.

164 __IDLOC directives must be contiguous
Do not place other code between __idloc directive declarations.
165 extended mode not available for this device
This PIC18 device does not support extended mode.

166 left bracket missing from offset operand
The left bracket is missing from an offset, i.e., [0x55.

167 right bracket missing from offset operand
The right bracket is missing from an offset, i.e., 0x55].

168 square brackets required around offset operand
Square brackets are required around an offset, i.e., [0x55]

169 access bit cannot be specified with indexed mode
When using indexed mode, the access bit cannot be specified.

170 expression within brackets must be constant
The expression specified within brackets is not a constant value.

171 address specified is not in access ram range of [0x60, 0xFF]
When making use of Access RAM, addressing must occur within the specified Access
Bank range.

172 PCL, TOSL, TOSH, or TOSU cannot be destination of MOVFF or
MOVSF
These registers cannot be written to with movff or movsf commands.

174 __CONFIG directives must be listed in ascending order
List config directive configuration registers in ascending order, i.e.,

    __CONFIG  _CONFIG0, _CP_OFF_0
    __CONFIG  _CONFIG1, _OSCS_OFF_1 & _RCIO_OSC_1
    __CONFIG  _CONFIG2, _BOR_ON_2 & _BORV_25_2

175 __IDLOCS directives must be listed in ascending order
List idlocs directive ID registers in ascending order, i.e.,

    __idlocs  _IDLOC0, 0x1
    __idlocs  _IDLOC1, 0x2
    __idlocs  _IDLOC2, 0x3

176 CONFIG Directive Error:
An error was found in the config directive syntax.

177 __CONFIG directives cannot be used with CONFIG directives
Do not mix __config directives and config directives when assigning configuration
bits in your code.

178 __CONFIG Directive Error:
An error was found in the __config directive syntax.
179 Instruction is not supported on this device
This error would occur when an instruction is used in code which is not supported on the particular family/architecture.

180 RES directive cannot reserve odd number of bytes in PIC18 absolute mode
This error would occur if you try to reserve an odd number of bytes using a PIC18 device and assemble in absolute mode (Quickbuild). For example:

```
org 0x0
a res 1
end
```
If you try to Quickbuild the above PIC18 MCU program, you will see error 180 and warning 231.

### UNKNOWN ERROR
An internal application error has occurred. (### is the value of the last defined error plus 1.)
Contact your Microchip Field Application Engineer (FAE) or Microchip support if you cannot debug this error.

### 8.3 ASSEMBLER WARNINGS
MPASM assembler warnings are listed numerically below:

201 Symbol not previously defined.
The symbol being undefined was not previously defined.

202 Argument out of range. Least significant bits used.
Argument did not fit in the allocated space. For example, literals must be 8 bits.

203 Found opcode in column 1.
An opcode was found in column one, which is reserved for labels.

204 Found pseudo-op in column 1.
A pseudo-op was found in column one, which is reserved for labels.

205 Found directive in column 1.
A directive was found in column one, which is reserved for labels.

206 Found call to macro in column 1.
A macro call was found in column one, which is reserved for labels.

207 Found label after column 1.
A label was found after column one, which is often due to a misspelled opcode.

208 Label truncated at 32 characters.
Maximum label length is 32 characters.

209 Missing quote.
A text string or character was missing a quote. For example, DATA 'a.
210 Extra ","
An extra comma was found at the end of the line.

211 Extraneous arguments on the line.
Extra arguments were found on the line.

212 Expected (ENDIF)
Expected an endif statement, i.e., an if statement was used without an endif.

213 The EXTERN directive should only be used when making a .o file.
The extern directive only has meaning if an object file is being created. This warning has been superseded by Error 149.

214 Unmatched ( 
An unmatched parenthesis was found. The warning is used if the parenthesis is not used for indicating order of evaluation.

215 Processor superseded by command line. Verify processor symbol.
The processor was specified on the command line as well as in the source file. The command line has precedence.
If you are using MPLAB IDE with the assembly, set the device to match the source file from Configure>Select Device.

216 Radix superseded by command line.
The radix was specified on the command line as well as in the source file. The command line has precedence.

217 Hex file format specified on command line.
The hex file format was specified on the command line as well as in the source file. The command line has precedence.

218 Expected DEC, OCT, HEX. Will use HEX.
Bad radix specification.

219 Invalid RAM location specified.
If the __maxram and __badram directives are used, this warning flags use of any RAM locations declared as invalid by these directives. Note that the provided header files include __maxram and __badram for each processor.

220 Address exceeds maximum range for this processor.
A ROM location was specified that exceeds the processor's memory size.

221 Invalid message number.
The message number specified for displaying or hiding is not a valid message number.

222 Error messages cannot be disabled.
Error messages cannot be disabled with the errorlevel command.

223 Redefining processor
The selected processor is being reselected by the list or processor directive.
224 Use of this instruction is not recommended.
The instruction is being obsoleted and is not recommended for current use. However, it is still supported for legacy reasons.

225 Invalid label in operand
Operand was not a valid address. For example, if the user tried to issue a CALL to a MACRO name.

226 Destination address must be word aligned
The destination address is not aligned with the start of a program memory word. For this device, use even bytes to specify address.

227 Substituting RETLW 0 for RETURN pseudo-op
Using `retlw 0` instead of `return` to resume program execution.

228 Invalid ROM location specified
The data memory location specified is not valid for the operation specified or is non-existent.

229 extended mode is not in effect -- overridden by command line
A command-line option has disabled extended mode operation.

230 __CONFIG has been deprecated for PIC18 devices. Use directive `CONFIG`.
Although you may still use the `__config` directive for PIC18 MCU devices, it is strongly recommended that you use the `config` directive (no leading underscores) instead. For PIC18FXXJ MCUs, you must use the `config` directive.

231 No memory has been reserved by this instruction
This warning would appear if an instruction which is meant to reserve memory cannot actually reserve that memory. For example:
```
org 0x0
a res 1
end
```
The above PIC18 assembly program is attempting to reserve one byte (`a res 1`) but this is not valid for a PIC18 MCU as each word size is two bytes.

232 STATUS register has no IRP or RP1 or RP0 bits
For PIC16 extended instruction devices, you are trying to access a non-existent bit of the STATUS register (IRP or RP1 or RP0 bits).

### UNKNOWN WARNING
An internal application error has occurred. (### is the value of the last defined warning plus 1.)
However, it is not severe enough to keep your code from assembling, i.e., it is a warning, not an error.
8.4 ASSEMBLER MESSAGES

MPASM assembler messages are listed numerically below:

301 MESSAGE:
User-definable message, invoked with the messg directive (see Section 4.48 "messg - Create User Defined Message").

302 Register in operand not in bank 0. Ensure that bank bits are correct.
This is a commonly seen reminder message to tell you that a variable that is being accessed in not in bank 0. This message was added to remind you to check your code, particularly code in banks other than 0. Review the section on banksel (Section 4.7 “banksel - Generate Bank Selecting Code”) and bankisel (Section 4.6 “bankisel - Generate Indirect Bank Selecting Code (PIC12/16 MCUs)”) and ensure that your code uses bank bits whenever changing from ANY bank to ANY other bank (including bank 0).
Since the assembler or linker can't tell which path your code will take, you will always get this message for any variable not in bank 0. You can use the errorlevel command to turn this and other messages on and off, but be careful as you may not spot a banking problem with this message turned off. For more about errorlevel, see Section 4.30 “errorlevel - Set Message Level”.
A similar message is 306 for paging.

303 Program word too large. Truncated to core size.
The program word (instruction width) is too large for the selected device’s core (program memory) size. Therefore the word has been truncated to the proper size.
For example, a 14-bit instruction would be truncated to 12 bits to be used by a PIC16F54.

304 ID Locations value too large. Last four hex digits used.
Only four hex digits are allowed for the ID locations.

305 Using default destination of 1 (file).
If no destination bit is specified, the default is used. Usually code that causes this message is missing the W or P after the register name, but sometimes the bug is due to typing movf instead of movwf.
It is best to fix any code that is causing this message. The default destination could not be where you want the value stored, and could cause the code to operate strangely.

306 Crossing page boundary -- ensure page bits are set.
Generated code is crossing a page boundary. This is a reminder message to tell you that code is being directed to a label that is on a page other than page 0. It is not an error or warning, but a reminder to check your page bits. Use the pagesel directive (Section 4.53 “pagesel - Generate Page Selecting Code (PIC10/12/16 MCUs)”) before this point and remember to use another pagesel if returning to page 0.
The assembler can't tell what path your code will take, so this message is generated for any label in a page other than 0. You can use the errorlevel command to turn this and other messages on and off, but be careful as you may not spot a paging problem with this message turned off. For more about errorlevel, see Section 4.30 “errorlevel - Set Message Level”.
A similar message is 302 for banking.
307 Setting page bits.
Page bits are being set with the LCALL or LGOTO pseudo-op.

308 Warning level superseded by command line value.
The warning level was specified on the command line as well as in the source file. The command line has precedence.

309 Macro expansion superseded by command line.
Macro expansion was specified on the command line as well as in the source file. The command line has precedence.

310 Superseding current maximum RAM and RAM map.
The __maxram directive has been used previously.

311 Operand of HIGH operator was larger than H'FFFF'.
High byte of address returned by high directive was greater than 0xFFFF.

312 Page or Bank selection not needed for this device. No code generated.
If a device contains only one ROM page or RAM bank, no page or bank selection is required, and any pagesel, banksel, or bankisel directives will not generate any code.

313 CBLOCK constants will start with a value of 0.
If the first cblock in the source file has no starting value specified, this message will be generated.

314 LFSR instruction is not supported on some versions of the 18Cxx2 devices.
See message 315 for more information.

315 Please refer to Microchip document DS80058A for more details
A downloadable PDF of this document, PIC18CXX2 Silicon/Data Sheet Errata, is available from the Microchip website.

316 W Register modified.
The working (W) register has been modified

317 W Register not modified. BSF/BCF STATUS instructions used instead.
The working (W) register has not been modified

318 Superseding current maximum ROM and ROM map.
Operation will cause maximum ROM to be exceeded.

### UNKNOWN MESSAGE
An internal application error has occurred. (### is the value of the last defined message plus 1.)
However, it is not severe enough to keep your code from assembling, i.e., it is a message, not an error.
8.5 ASSEMBLER LIMITATIONS

8.5.1 General Limitations

- If a fully qualified path is specified, only that path will be searched. Otherwise, the search order is: (1) current working directory, (2) source file directory, and (3) MPASM assembler executable directory.
- There is a source file line limit (expanded) of 200 characters.

8.5.2 Directive Limitations

- Do not use \#includes in macros.
- if directive limits
  - Maximum nesting depth = 16
- include directive limits
  - Maximum nesting depth = 5
  - Maximum number of files = 255
- macro directive limits
  - Maximum nesting depth = 16
- while directive limits
  - Maximum nesting depth = 8
  - Maximum number of lines per loop = 100
  - Maximum iterations = 256

8.5.3 MPASM Assembler Versions before v3.30

Assembler versions before v3.30 (v3.2x and earlier) have limitations based on the generation a COD file for debugging and the support of a command-line version, mpasm.exe.

- There is a 62 character length restriction for file and path names in the debug (COD) file produced by MPASM assembler. This can cause problems when assembling single files with long file names and/or path names.
  Work arounds:
  - Shorten your file name or move your file into a directory closer to the root directory (shorten the path name), and try assembling your file again.
  - Create a mapped drive for the long directory chain.
  - Use the linker with the assembler, and not the assembler alone, to generate your output. There is no character restriction with MPLINK linker.
- The command-line version of the assembler (mpasm.exe) has the following limitations:
  - File names are limited to 8.3 format.
  - config directive not supported.
Part 2 – MPLINK Object Linker

Chapter 9. MPLINK Linker Overview ............................................................... 171
Chapter 10. Linker Interfaces .......................................................................... 179
Chapter 11. Linker Scripts ............................................................................... 181
Chapter 12. Linker Processing ......................................................................... 191
Chapter 13. Sample Applications ................................................................... 195
Chapter 14. Errors, Warnings and Common Problems .................................. 223
Chapter 9. MPLINK Linker Overview

9.1 INTRODUCTION

An overview of the MPLINK object linker and its capabilities is presented.
Topics covered in this chapter:
- MPLINK Linker Defined
- How MPLINK Linker Works
- How MPLINK Linker Helps You
- Linker Platforms Supported
- Linker Operation
- Linker Input/Output Files

9.2 MPLINK LINKER DEFINED

MPLINK object linker (the linker) combines object modules generated by the MPASM assembler or the MPLAB C18 C compiler into a single executable (hex) file. The linker also accepts libraries of object files as input, as generated by the MPLIB object librarian. The linking process is controlled by a linker script file, which is also input into MPLINK linker.

For more information on MPASM assembler, see Chapter 1. “MPASM Assembler Overview”. For more information on MPLAB C18, see C compiler documentation listed in Recommended Reading.

9.3 HOW MPLINK LINKER WORKS

MPLINK linker performs many functions:
- Locates Code and Data. The linker takes as input relocatable object files. Using the linker script, it decides where the code will be placed in program memory and where variables will be placed in RAM.
- Resolves Addresses. External references in a source file generate relocation entries in the object file. After the linker locates code and data, it uses this relocation information to update all external references with the actual addresses.
- Generates an Executable. Produces a .hex file that can be programmed into a PIC1X MCU or loaded into an emulator or simulator to be executed.
- Configures Stack Size and Location. Allows MPLAB C18 to set aside RAM space for dynamic stack usage.
- Identifies Address Conflicts. Checks to ensure that program/data do not get assigned to space that has already been assigned or reserved.
- Provides Symbolic Debug Information. Outputs a file that MPLAB IDE uses to track address labels, variable locations, and line/file information for source level debugging.
9.4 HOW MPLINK LINKER HELPS YOU

MPLINK linker allows you to produce modular, reusable code. Control over the linking process is accomplished through a linker script file and with command line options. The linker ensures that all symbolic references are resolved and that code and data fit into the available PIC1X MCU device.

MPLINK linker can help you with:

- Reusable Source Code. You can build up your application in small, reusable modules.
- Libraries. You can make libraries of related functions which can be used in creating efficient, readily compilable applications.
- MPLAB C18. The Microchip compiler for PIC18 devices requires the use of MPLINK linker and can be used with precompiled libraries and MPASM assembler.
- Centralized Memory Allocation. By using application-specific linker scripts, precompiled objects and libraries can be combined with new source modules and placed efficiently into available memory at link time.
- Accelerated Development. Since tested modules and libraries don't have to be recompiled each time a change is made in your code, compilation time may be reduced.

9.5 LINKER PLATFORMS SUPPORTED

MPLINK linker is distributed as a Windows 32 console application suitable for Windows 95/98 and Windows NT/2000/XP platforms.

9.6 LINKER OPERATION

The MPLINK linker combines multiple input object modules and library files, per the linker script file, into a single output COF file. Utilities can be used to generate executable code (.hex) or a linker listing file (.lst) from the COF file. A map file can also be generated to aid in debugging.

FIGURE 9-1: MPLINK™ LINKER OPERATION

* The linker can select this file for you.
The linker is executed after assembling or compiling relocatable object modules with the MPASM assembler and/or MPLAB C18 C compiler. The actual addresses of data and the location of functions will be assigned when the MPLINK linker is executed. This means that you may instruct the linker, via a linker script, to place code and data somewhere within named regions of memory, or, if not specified, to place into any available memory.

The linker script must also tell the MPLINK linker about the ROM and RAM memory regions available in the target PIC1X MCU device. Then, it can analyze all the input files and try to fit the application's routines into ROM and assign its data variables into available RAM. If there is too much code or too many variables to fit, the linker will give an error message.

The MPLINK linker also provides flexibility for specifying that certain blocks of data memory are reusable, so that different routines (which never call each other and which don’t depend upon this data to be retained between execution) can share limited RAM space.

When using a C compiler, libraries are available for most PIC MCU peripheral functions as well as for many standard C functions. The linker will only extract and link individual object files that are needed for the current application from the included libraries. This means that relatively large libraries can be used in a highly efficient manner.

The MPLINK linker combines all input files and ensure that all addresses are resolved. Any function in the various input modules that attempts to access data or call a routine that has not been allocated or created will cause the linker to generate an error.

Finally the linker calls the MP2HEX utility to generate the executable output. The MPLINK linker also generates symbolic information for debugging your application with MPLAB IDE (.cof and .map files). A list file (.lst) can also be generated by calling the MP2COD utility.

### 9.7 LINKER INPUT/OUTPUT FILES

The MPLINK linker combines multiple object files into one executable hex file.

#### Input Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object File (.o)</td>
<td>Relocatable code produced from a source file.</td>
</tr>
<tr>
<td>Library File (.lib)</td>
<td>A collection of object files grouped together for convenience.</td>
</tr>
<tr>
<td>Linker Script File (.lkr)</td>
<td>Description of memory layout for a particular processor/project.</td>
</tr>
</tbody>
</table>

#### Output Files

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COFF Object Module File (.cof, .out)</td>
<td>Debug file used by MPLAB IDE v6.xx and later.</td>
</tr>
<tr>
<td>Hex File Formats (.hex, .hxl, .hxh)</td>
<td>Hexadecimal file with no debug information. Suitable for use in programming. This file is generated by the utility MP2HEX.</td>
</tr>
<tr>
<td>Listing File (.lst)</td>
<td>Original source code, side-by-side with final binary code. <strong>Note:</strong> Requires linker can find original source files. This file is generated by the utility MP2COD.</td>
</tr>
<tr>
<td>Map File (.map)</td>
<td>Shows the memory layout after linking. Indicates used and unused memory regions.</td>
</tr>
</tbody>
</table>
9.7.1 Object File (.o)

Object files are the relocatable code produced from source files. The MPLINK linker combines object files and library files, according to a linker script, into a single output file.

Object files may be created from source files by MPASM assembler and library files may be created from object files by MPLIB librarian.

9.7.2 Library File (.lib)

Libraries are a convenient way of grouping related object modules. A library file may be created from object files by MPLIB librarian. For more on the librarian, see Chapter 15. “MPLIB Librarian Overview”.

9.7.3 Linker Script File (.lkr)

Linker script files are the command files of MPLINK linker. For more information on linker scripts, see Chapter 11. “Linker Scripts”.

Standard linker script files are located in:
C:\Program Files\Microchip\MPASM Suite\LKR

During the link process, if MPLINK linker is unable to resolve a reference to a symbol, it will search libraries specified on the command line or in the linker script in an attempt to resolve the reference. If a definition is found in a library file, the object file containing that definition will be included in the link.

9.7.4 COFF Object Module File (.cof, .out)

MPLINK linker generates a COFF file which provides debugging information to MPLAB IDE v6.xx or later.

9.7.5 Hex File Formats (.hex, .hxl, .hxh)

Both the MPASM assembler and the MPLINK linker can generate a hex file. For more information on this format, see Section 1.7.5 “Hex File Formats (.hex, .hxl, .hxh)”. For MPLINK linker, mp2hex.exe uses the COF file to generate the hex file. To prevent hex file generation, use the /x option.

9.7.6 Listing File (.lst)

An MPLINK linker listing file provides a mapping of source code to object code. It also provides a list of symbol values, memory usage information, and the number of errors, warnings and messages generated. This file may be viewed in MPLAB IDE by:

1. selecting File>Open to launch the Open dialog
2. selecting "List files (*.lst)" from the “Files of type” drop-down list
3. locating the desired list file
4. clicking on the list file name
5. clicking Open

Both the MPASM assembler and the MPLINK linker can generate listing files. For information on the MPASM assembler listing file, see Section 1.7.3 “Listing File (.lst)”.

An alternative to a listing file would be to use the information in the Disassembly window (View>Disassembly) in MPLAB IDE.

The MPLINK linker uses the mp2cod.exe utility to generate the linker list file from the COF file. To prevent linker list file generation, use the /w option.
EXAMPLE 9-1: MPLINK LINKER LISTING FILE

The MP2COD utility version and list file generation data appear at the top of each page. The first column contains the base address in memory where the code will be placed. The second column is reserved for the machine instruction. This is the code that will be executed by the PIC MCU. The third column displays disassembly code. The fourth column lists the associated source code line. The fifth column lists the file associated for the source code line.

Note: Due to page width restrictions, some comments have been shortened, indicated by ".." Also, associated file names have been replaced by numbers, i.e., (1) and (2). See the end of the listing of the actual file paths and names.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Disassembly</th>
<th>Source</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------</td>
<td>-----</td>
<td>------------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>include p18f452.inc (1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LIST (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>; P18F452.INC Standard Header File,... (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LIST (2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>udata (1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Dest res 1 (1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RST code 0x0 (1)</td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>ef16</td>
<td>GOTO 0x2c</td>
<td>goto Start (1)</td>
<td></td>
</tr>
<tr>
<td>000002</td>
<td>f000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PGM code</td>
<td></td>
</tr>
<tr>
<td>00002c</td>
<td>0e0a</td>
<td>MOVLL 0xa</td>
<td>movlw 0x0A (1)</td>
<td></td>
</tr>
<tr>
<td>00002e</td>
<td>6f80</td>
<td>MOVWF 0x80,0x1</td>
<td>movwf Dest (1)</td>
<td></td>
</tr>
<tr>
<td>000030</td>
<td>9780</td>
<td>BCF 0x80,0x3,0x1</td>
<td>bcf Dest, 3 (1)</td>
<td></td>
</tr>
<tr>
<td>000032</td>
<td>ef16</td>
<td>GOTO 0x2c</td>
<td>goto Start (1)</td>
<td></td>
</tr>
<tr>
<td>000034</td>
<td>f000</td>
<td></td>
<td>end (1)</td>
<td></td>
</tr>
</tbody>
</table>

where:

(1) = D:\Projects32\PIC18F452\SourceReloc.asm
(2) = C:\Program Files\Microchip\MPASM Suite\p18f452.inc
9.7.7 Map File (.map)

The map file generated by MPLINK linker can be viewed by selecting File>Open in MPLAB IDE and choosing the file you specified in the MPLINK linker options. It provides information on the absolute location of source code symbols in the final output. It also provides information on memory use, indicating used/unused memory. This window is automatically reloaded after each rebuild.

The map file contains four tables. The first table (Section Info) displays information about each section. The information includes the name of the section, its type, beginning address, whether the section resides in program or data memory, and its size in bytes.

There are four types of sections:
- code
- initialized data (idata)
- uninitialized data (udata)
- initialized ROM data (romdata)

The following table is an example of the section table in a map file:

<table>
<thead>
<tr>
<th>Section</th>
<th>Type</th>
<th>Address</th>
<th>Location</th>
<th>Size(Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>code</td>
<td>0x000000</td>
<td>program</td>
<td>0x000002</td>
</tr>
<tr>
<td>.cinit</td>
<td>romdata</td>
<td>0x000021</td>
<td>program</td>
<td>0x000004</td>
</tr>
<tr>
<td>.code</td>
<td>code</td>
<td>0x000023</td>
<td>program</td>
<td>0x000026</td>
</tr>
<tr>
<td>.udata</td>
<td>udata</td>
<td>0x000020</td>
<td>data</td>
<td>0x000005</td>
</tr>
</tbody>
</table>

The second table (Program Memory Usage) lists program memory addresses that were used and provides a total usage statistic. For example:

<table>
<thead>
<tr>
<th>Program Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
</tr>
<tr>
<td>0x000000</td>
</tr>
<tr>
<td>0x00002a</td>
</tr>
<tr>
<td>0x0000bc</td>
</tr>
<tr>
<td>0x001176</td>
</tr>
</tbody>
</table>

10209 out of 32786 program addresses used, program memory utilization is 31%

The third table in the map file (Symbols - Sorted by Name) provides information about the symbols in the output module. The table is sorted by the symbol name and includes the address of the symbol, whether the symbol resides in program or data memory, whether the symbol has external or static linkage, and the name of the file where defined. The following table is an example of the symbol table sorted by symbol name in a map file:

<table>
<thead>
<tr>
<th>Symbols - Sorted by Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>call_m</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>main</td>
</tr>
<tr>
<td>mpy</td>
</tr>
<tr>
<td>start</td>
</tr>
<tr>
<td>H_byte</td>
</tr>
<tr>
<td>L_byte</td>
</tr>
<tr>
<td>count</td>
</tr>
<tr>
<td>mulcnd</td>
</tr>
<tr>
<td>muplr</td>
</tr>
</tbody>
</table>
The fourth table in the map file (Symbols - Sorted by Address) provides the same information that the third table provides, but it is sorted by symbol address rather than symbol name.

If a linker error is generated, a complete map file can not be created. However, if the /m option was supplied, an error map file will be created. The error map file contains only section information; no symbol information is provided. The error map file lists all sections that were successfully allocated when the error occurred. This file, in conjunction with the error message, should provide enough context to determine why a section could not be allocated.
Chapter 10. Linker Interfaces

10.1 INTRODUCTION

MPLINK object linker usage is discussed.

When MPLAB IDE or MPLAB C18 is installed, the MPLINK linker (mplink.exe) is also installed.

Topics covered in this chapter:
- MPLAB IDE Interface
- Command Line Interface
- Command Line Example

10.2 MPLAB IDE INTERFACE

The MPLINK linker is commonly used with the MPASM assembler in an MPLAB IDE project to generate relocatable code. For more information on this use, see “PIC1X MCU Language Tools and MPLAB IDE”.

The linker may also be used in MPLAB IDE with the MPLAB C18 C compiler. For more information on Microchip compilers, see the MPLAB C18 C compiler documentation listed in Recommended Reading.

10.3 COMMAND LINE INTERFACE

MPLINK linker can be used in MPLAB IDE or directly from a command line.

When used in MPLAB IDE, all of MPLINK linker’s options are available through the MPLINK Linker tab, accessed from the Project>Build Options dialog.

When using MPLINK linker in a batch file, or directly from the command line, the linker is invoked with the following two syntaxes:

```
mplink cmdfiles objfiles [libfiles] [options]
mplink /ppartnumber objfiles [libfiles] [options]
```

*cmdfile* is the name of a linker command file. All linker command files must have the extension .lkr.

*partnumber* indicates the part number for the PIC MCU’s generic linker script to build the project. The linker will search the lkr directory to find the generic linker script for that part. The lkr directory is located at the same location as the MPLINK linker executable. The linker will construct the name of the generic linker script by adding an ‘_g.lkr’ to the string value of the part number. For example, the generic linker script for PIC18F4520 is 18f4520_g.lkr.

*objfile* is the name of an assembler or compiler generated object file. All object files must have the extension .o.

*libfile* is the name of a librarian-created library file. All library files must have the extension .lib.
option is one of the linker command-line options described below.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/a hexformat</td>
<td>Specify format of hex output file.</td>
</tr>
<tr>
<td>/h, /?</td>
<td>Display help screen.</td>
</tr>
<tr>
<td>/k pathlist</td>
<td>Add directories to linker script search path.</td>
</tr>
<tr>
<td>/l pathlist</td>
<td>Add directories to library search path.</td>
</tr>
<tr>
<td>/m filename</td>
<td>Create map file filename.</td>
</tr>
<tr>
<td>/n length</td>
<td>Specify number of lines per listing page.</td>
</tr>
<tr>
<td>/o filename</td>
<td>Specify output file filename. Default is a.out.</td>
</tr>
<tr>
<td>/q</td>
<td>Quiet mode.</td>
</tr>
<tr>
<td>/u</td>
<td>Specify multiple macros using following syntax:</td>
</tr>
<tr>
<td></td>
<td>/u &lt;sym[=value]&gt;</td>
</tr>
<tr>
<td></td>
<td>where sym may be a macro with alphanumeric characters</td>
</tr>
<tr>
<td></td>
<td>and value maybe be a numerical value.</td>
</tr>
<tr>
<td></td>
<td>If a value is not provided, 0 will be used.</td>
</tr>
<tr>
<td>/w</td>
<td>Suppress the mp2cod.exe utility. Using this option</td>
</tr>
<tr>
<td></td>
<td>will prevent the generation of a .lst file.</td>
</tr>
<tr>
<td>/x</td>
<td>Suppress the mp2hex.exe utility. Using this option</td>
</tr>
<tr>
<td></td>
<td>will prevent the generation of a .hex file.</td>
</tr>
</tbody>
</table>

There is no required order for the command line arguments; however, changing the order can affect the operation of the linker. Specifically, additions to the library/object directory search path are appended to the end of the current library/object directory search path as they are encountered on the command line and in command files.

Library and object files are searched for in the order in which directories occur in the library/object directory search path. Therefore, changing the order of directories may change which file is selected.

The /o option is used to supply the name of the generated output COFF file for MPLAB IDE debugging. Also generated is an Intel format hex file for programming. This file has the same name as the output COFF file but with the file extension .hex. If the /o option is not supplied, the default output COFF file is named a.out and the corresponding hex file is named a.hex.

### 10.4 COMMAND LINE EXAMPLE

An example of an MPLINK linker command line is shown below.

```bash
mplink 18f452.lkr main.o funct.o math.lib /m main.map /o main.out
```

This instructs MPLINK linker to use the 18f452.lkr linker script file to link the input modules main.o, funct.o, and the precompiled library math.lib. It also instructs the linker to produce a map file named main.map, main.o and funct.o must have been previously compiled or assembled. The output files main.cof and main.hex will be produced if no errors occur during the link process.
Chapter 11. Linker Scripts

11.1 INTRODUCTION

Linker script files are used by the linker to generate application code. You no longer need to add a device-specific linker script file to the command line or your MPLAB IDE project; the linker will find the appropriate file for you as long as a device has been specified. However, if you want to use a non-standard linker script file, you will have to add that manually.

Depending on the hardware debug tool you want to use, you may need to set certain conditional symbols on the command line (see Example 11.8.4) or to select the Build Configuration as “Debug” in MPLAB IDE.

Linker script files are the command files of the linker. They specify:
• Program and data memory regions for the target part
• Stack size and location (for MPLAB C18)
• A mapping of logical sections in source code into program and data regions

Linker script directives form the command language that controls the linker's behavior. There are four basic categories of linker script directives. Each of these directives, plus some useful linker script caveats, are discussed in the topics listed below.

Note: Linker script comments are specified by '//', i.e., any text between a '//' and the end of a line is ignored.

Topics covered in this chapter:
• Standard Linker Scripts
• Linker Script Command Line Information
• Linker Script Caveats
• Memory Region Definition
• Logical Section Definition
• STACK Definition
• Conditional Linker Statements

11.2 STANDARD LINKER SCRIPTS

Standard linker script files are provided for each device and are located, by default, in the directory: C:\Program Files\Microchip\MPASM Suite\LKR.

Standard linker scripts are named with the following convention partnumber_g.lkr. For example, the standard linker script for PIC16F872 is 16F872_g.lkr. The standard linker scripts contain conditional linker statements and MPLAB IDE uses the /u command line flag to utilize these statements for different builds such as debug or no debug. You can modify a local copy of the standard linker script and use it in your project.
11.3 LINKER SCRIPT COMMAND LINE INFORMATION

The MPLAB IDE Project Manager can set this information directly. You probably only need to use these if you are linking from the command line.

- LIBPATH
- LKRPATH
- FILES
- INCLUDE

11.3.1 LIBPATH

Library and object files which do not have a path are searched using the library/object search path. The following directive appends additional search directories to the library/object search path:

LIBPATH libpath

where libpath is a semicolon-delimited list of directories.

EXAMPLE 11-1: LIBPATH EXAMPLE

To append the current directory and the directory C:\PROJECTS\INCLUDE to the library/object search path, the following line should be added to the linker command file:

LIBPATH .;C:\PROJECTS\INCLUDE

11.3.2 LKRPATH

Linker command files that are included using a linker script INCLUDE directive are searched for using the linker command file search path. The following directive appends additional search directories to the linker command file search path:

LKRPATH lkrpath

where lkrpath is a semicolon-delimited list of directories.

EXAMPLE 11-2: LKRPATH EXAMPLE

To append the current directory's parent and the directory C:\PROJECTS\SCRIPTS to the linker command file search path, the following line should be added to the linker command file:

LKRPATH ..;C:\PROJECTS\SCRIPTS

11.3.3 FILES

The following directive specifies object or library files for linking:

FILES objfile/libfile [objfile/libfile...]

where objfile/libfile is either an object or library file.

Note: More than one object or library file can be specified in a single FILES directive.

EXAMPLE 11-3: FILES EXAMPLE

To specify that the object module main.o be linked with the library file math.lib, the following line should be added to the linker command file:

FILES main.o math.lib
11.3.4 INCLUDE

The following directive includes an additional linker command file:

```
INCLUDE cmdfile
```

where `cmdfile` is the name of the linker command file to include.

**EXAMPLE 11-4: INCLUDE EXAMPLE**

To include the linker command file named `mylink.lkr`, the following line should be added to the linker command file:

```
INCLUDE mylink.lkr
```

11.4 LINKER SCRIPT CAVEATS

Some linker script caveats:

- You may need to modify the linker script files included with MPLINK linker before using them.
- You may wish to reconfigure stack size to use MPLAB C18 with MPLINK linker.
- You will need to split up memory pages if your code contains `goto` or `call` instructions without `pagesel` pseudo-instructions (directives.)
- You must not combine data memory regions when using MPLINK linker with MPLAB C18 C compiler. MPLAB C18 requires that any section be located within a single bank. See MPLAB C18 documentation for directions on creating variables larger than a single bank.

11.5 MEMORY REGION DEFINITION

The linker script describes the memory architecture of the PIC1X MCU. This allows the linker to place code in available ROM space and variables in available RAM space. Regions that are marked `PROTECTED` will not be used for general allocation of program or data. Code or data will only be allocated into these regions if an absolute address is specified for the section, or if the section is assigned to the region using a `SECTION` directive in the linker script file.

11.5.1 Defining RAM Memory Regions

The `DATABANK`, `SHAREBANK` and `ACCESSBANK` directives are used for variable data in internal RAM. The formats for these directives are as follows.

**Banked Registers**

```
DATABANK NAME=memName START=addr END=addr [PROTECTED]
```

**Unbanked Registers**

```
SHAREBANK NAME=memName START=addr END=addr [PROTECTED]
```

**Access Registers (PIC18 devices only)**

```
ACCESSBANK NAME=memName START=addr END=addr [PROTECTED]
```

where:

- `memName` is any ASCII string used to identify an area in RAM.
- `addr` is a decimal (e.g., .30) or hexadecimal (e.g., 0xFF) number specifying an address.

The optional keyword `PROTECTED` indicates a region of memory that only can be used when specifically identified in the source code. The linker will not use the protected area.
EXAMPLE 11-5:  RAM EXAMPLE

Based on the RAM memory layout shown in PIC16F877A Register File Map, the DATABANK and SHAREBANK entries in the linker script file would appear as shown in the examples below the map.

PIC16F877A Register File Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>INDF0</td>
<td>INDF0</td>
<td>INDF0</td>
<td>INDF0</td>
</tr>
<tr>
<td>01h</td>
<td>TMR0</td>
<td>OPTION_REG</td>
<td>TMR0</td>
<td>OPTION_REG</td>
</tr>
<tr>
<td>02h</td>
<td>PCL</td>
<td>PCL</td>
<td>PCL</td>
<td>PCL</td>
</tr>
<tr>
<td>03h</td>
<td>STATUS</td>
<td>STATUS</td>
<td>STATUS</td>
<td>STATUS</td>
</tr>
<tr>
<td>04h</td>
<td>FSR</td>
<td>FSR</td>
<td>FSR</td>
<td>FSR</td>
</tr>
<tr>
<td>05h</td>
<td>PORTA</td>
<td>TRISA</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>INDF1</td>
<td>INDF1</td>
<td>INDF1</td>
<td>INDF1</td>
</tr>
<tr>
<td></td>
<td>TMR1</td>
<td>TMR1</td>
<td>TMR1</td>
<td>TMR1</td>
</tr>
<tr>
<td>0Fh</td>
<td>TMR1H</td>
<td>—</td>
<td>EEADRH</td>
<td>—</td>
</tr>
<tr>
<td>10h</td>
<td>T1CON</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>11h</td>
<td>ADRCN1</td>
<td>ADRCN1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>12h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>13h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>14h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>15h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>16h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>17h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>18h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>19h</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1Ah</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1Bh</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1Ch</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1Dh</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1 Eh</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>1Fh</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

RAM Memory Declarations for PIC16F877A - Banked Memory

//Special Function Registers in Banks 0-3
DATABANK NAME=sfr0 START=0x0 END=0x1F PROTECTED
DATABANK NAME=sfr1 START=0x80 END=0x9F PROTECTED
DATABANK NAME=sfr2 START=0x100 END=0x10F PROTECTED
DATABANK NAME=sfr3 START=0x180 END=0x18F PROTECTED

//General Purpose RAM in Banks 0-3
DATABANK NAME=gpr0 START=0x20 END=0x6F
DATABANK NAME=gpr1 START=0xA0 END=0xEF
DATABANK NAME=gpr2 START=0x110 END=0x16F
DATABANK NAME=gpr3 START=0x190 END=0x1BF

RAM Memory Declarations for PIC16F877A - Unbanked Memory

//General Purpose RAM - available in all banks
SHAREBANK NAME=gprncbknk START=0x70 END=0x7F
SHAREBANK NAME=gprncbknk START=0xF0 END=0xFF
SHAREBANK NAME=gprncbknk START=0x170 END=0x17F
SHAREBANK NAME=gprncbknk START=0x1F0 END=0x1FF
11.5.2 Defining ROM Memory Regions

The CODEPAGE directive is used for program code, initialized data values, constant data values and external memory. It has the following format:

```
CODEPAGE NAME=memName START=addr END=addr [PROTECTED] [FILL=fillvalue]
```

where:

- `memName` is any ASCII string used to identify a CODEPAGE.
- `addr` is a decimal or hexadecimal number specifying an address.
- `fillValue` is a value which fills any unused portion of a memory block. If this value is in decimal notation, it is assumed to be a 16-bit quantity. If it is in hexadecimal notation (e.g., 0x2346), it may be any length divisible by full words (16 bits).

The optional keyword `PROTECTED` indicates a region of memory that only can be used by program code that specifically requests it.

EXAMPLE 11-6: ROM EXAMPLE

The program memory layout for a PIC16F877A microcontroller is shown below.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset Vector</td>
<td>Start: 0000h</td>
</tr>
<tr>
<td>Interrupt Vector</td>
<td>Start: 0004h</td>
</tr>
<tr>
<td>User Memory Space</td>
<td>0005h - 07FFh</td>
</tr>
<tr>
<td>User Memory Space</td>
<td>0800h - 0FFFh</td>
</tr>
<tr>
<td>User Memory Space</td>
<td>1000h - 17FFh</td>
</tr>
<tr>
<td>User Memory Space</td>
<td>1800h - 1FFFh</td>
</tr>
<tr>
<td>ID Locations</td>
<td>2000h - 2003h</td>
</tr>
<tr>
<td>Reserved</td>
<td>2004h - 2005h</td>
</tr>
<tr>
<td>Device ID</td>
<td>2006h</td>
</tr>
<tr>
<td>Configuration Memory Space</td>
<td>2007h</td>
</tr>
<tr>
<td>Reserved</td>
<td>2008h - 20FFh</td>
</tr>
<tr>
<td>EEPROM Data</td>
<td>2100h - 21FFh</td>
</tr>
</tbody>
</table>

Based on this map, the CODEPAGE declarations are:

```
CODEPAGE NAME=page0 START=0x0000 END=0x07FF
CODEPAGE NAME=page1 START=0x0800 END=0x0FFF
CODEPAGE NAME=page2 START=0x1000 END=0x17FF
CODEPAGE NAME=page3 START=0x1800 END=0x1FFF
CODEPAGE NAME=.idlocs START=0x2000 END=0x2003 PROTECTED
CODEPAGE NAME=.config START=0x2007 END=0x2007 PROTECTED
CODEPAGE NAME=eedata START=0x2100 END=0x21FF PROTECTED
```
11.6 LOGICAL SECTION DEFINITION

Logical sections are used to specify which of the defined memory regions should be used for a portion of source code. To use logical sections, define the section in the linker script file with the `SECTION` directive and then reference that name in the source file using that language's built-in mechanism (e.g., `#pragma section` for MPLAB C18).

The section directive defines a section by specifying its name, and either the block of program memory in ROM or the block of data memory in RAM which contains the section:

```
SECTION NAME=secName { ROM=memName | RAM=memName }
```

where:
- `secName` is an ASCII string used to identify a section.
- `memName` is a previously defined `ACCESSBANK`, `SHAREBANK`, `DATABANK`, or `CODEPAGE`.

The `ROM` attribute must always refer to program memory previously defined using a `CODEPAGE` directive. The `RAM` attribute must always refer to data memory previously defined with a `ACCESSBANK`, `DATABANK` or `SHAREBANK` directive.

**EXAMPLE 11-7: LOGICAL SECTION DEFINITION**

To specify that a section whose name is `filter_coeffs` be loaded into the region of program memory named `constants`, the following line should be added to the linker command file:

```
SECTION NAME=filter_coeffs ROM=constants
```

**EXAMPLE 11-8: LOGICAL SECTION USAGE**

To place MPASM source code into a section named `filter_coeffs`, use the following line prior to the desired source code:

```
filter_coeffs CODE
```

11.7 STACK DEFINITION

Only MPLAB C18 requires a software stack be set up. The following statement specifies the stack size and an optional DATABANK where the stack is to be allocated:

```
STACK SIZE=allocSize [RAM=memName]
```

where:
- `allocSize` is the size in bytes of the stack and `memName` is the name of a memory previously declared using a `ACCESSBANK`, `DATABANK` or `SHAREBANK` statement.

**EXAMPLE 11-9: STACK EXAMPLE**

To set the stack size to be 0x20 in the RAM area previously defined by `gpr0`, the following line should be added to the linker command file:

```
STACK SIZE=0x20 RAM=gpr0
```
11.8 CONDITIONAL LINKER STATEMENTS

Generic linker scripts contain conditional statements and macros to accommodate several different methods for linking code:

- Debug vs. Release (e.g., for the MPLAB REAL ICE™ in-circuit emulator)
- C code vs. Assembly
- PIC18 Extended Microcontroller mode vs. Traditional mode

Being able to use one linker script instead of several simplifies application development.

MPLINK linker accepts IF/ELSE type conditional statements in the linker scripts, as discussed below. Several macros are used in support of the conditional statements. Also, certain directives are useful with these conditional statements.

11.8.1 #IFDEF/ELSE/FI

Two syntaxes are accepted for these conditional statements:

**Conditional 1**

```
#IFDEF
....
#FI
```

**Conditional 2**

```
#IFDEF
....
#ELSE
....
#FI
```

11.8.1.1 #IFDEF

Only one macro is allowed after this directive. If the macro is defined before, the if clause will be parsed by the linker. Complex conditions must be constructed using nested if-else clauses.

11.8.1.2 #ELSE

No macro is allowed after this directive. The else clause will be parsed only in case that the if clause is not.

11.8.1.3 #FI

No macro is allowed after this directive. It identifies the end of if or else clause.

11.8.2 Macros

Depending on what you want to do, macros may be set and used with conditional statements to determine how the linker script will be interpreted by the linker.

If you are using MPLAB IDE, all macros listed in Table 11-1 will automatically be set for you. The only exception is Debug vs. Release, which must be selected under the Build Configuration. See MPLAB IDE documentation for more on how to set the Build Configuration.
If you are using the command line, you must set the macros yourself. The macros that are available for you to set are listed below. On the command line, precede the macro with /u. See Section 11.8.4 “Examples of Use” for some examples.

### TABLE 11-1: LINKER SCRIPT MACROS

<table>
<thead>
<tr>
<th>Macro</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CRUNTIME</td>
<td>To link C code or mixed C code and assembly.</td>
</tr>
<tr>
<td>_EXTENDEDMODE</td>
<td>To use PIC18 extended microcontroller mode.</td>
</tr>
<tr>
<td>_DEBUG</td>
<td>To specify debug mode, as opposed to release, or production, mode.</td>
</tr>
<tr>
<td>_DEBUGCODESTART</td>
<td>To set the start in program memory of the debug executive; i.e.,</td>
</tr>
<tr>
<td></td>
<td>/u_DEBUGCODESTART=address</td>
</tr>
<tr>
<td>_DEBUGCODELEN</td>
<td>To set the size of the debug executive; i.e.,</td>
</tr>
<tr>
<td></td>
<td>/u_DEBUGCODELEN=hexvalue</td>
</tr>
<tr>
<td>_DEBUGDATASTART</td>
<td>To set the start of data memory reserved registers; i.e.,</td>
</tr>
<tr>
<td></td>
<td>/u_DEBUGDATASTART=address</td>
</tr>
<tr>
<td>_DEBUGDATALEN</td>
<td>To set the amount of data memory reserved; i.e.,</td>
</tr>
<tr>
<td></td>
<td>/u_DEBUGDATALEN=hexvalue</td>
</tr>
</tbody>
</table>

### 11.8.3 Supporting Directives

The #DEFINE directive may be used to define a macro or define it and set its value. The ERROR directive can be used within an if-else clause.

#### 11.8.3.1 #DEFINE

Through this directive, you can define a macro and associate a numerical value to it. The value can only be calculated using an '+', '-', '/' or '*' operator over two previously defined macros. Complex calculations must be constructed using combination of multiple #DEFINE directives.

The following syntaxes are accepted:

- `#DEFINE newmacromacro1 + macro2`
- `#DEFINE newmacromacro1 - macro2`
- `#DEFINE newmacromacro1 / macro2`
- `#DEFINE newmacromacro1 * macro2`

*newmacro* may not be a previously defined macro.

*macro1* and *macro2* are previously defined macros. The numerical values associated to these macros will be used to calculate a numerical value for *newmacro*.

#### 11.8.3.2 ERROR

An ERROR directive has been added to MPLINK linker. This directive allows you to stop the linker and emit the message in front of it on the standard output. The syntax of this directive is:

```
ERROR msg
```

This directive is used in the generic linker scripts to calculate the begin and end of the debug sections in code and data memory.
11.8.4 Examples of Use

In these examples, the generic linker script for PIC18F6722 is used to demonstrate how different options for building a project can be selected.

1. Building a C project, Extended mode:
   mplink.exe /p18F6722 /u_CRUNTIME /u_EXTENDEDMODE <other flags>

2. Building a C project for PIC18F6722 with debug sections for code at 0x1fd80 and for data at 0xef4, Traditional (non-extended) mode:
   mplink.exe /p18F6722 /u_CRUNTIME /u_DEBUG /u_DEBUGCODESTART=0x1fd80 /u_DEBUGCODELEN=0x280 /u_DEBUGDATATART=0xef4 /u_DEBUGDATALEN=0xc <other flags>

3. Building a Assembly project, no debug:
   mplink.exe /p18f6722 <other flags>

Generic Linker Script - 18f6722_g.lkr

// File: 18f6722_g.lkr
// Generic linker script for the PIC18F6722 processor

#DEFINE _CODEEND _DEBUGCODESTART - 1
#DEFINE _CEND _DEBUGCODESTART + _DEBUGCODELEN
#DEFINE _DATAEND _DEBUGDATASTART - 1
#DEFINE _DEND _DEBUGDATASTART + _DEBUGDATALEN

LIBPATH .

#if _CRUNTIME
    #ifdef _EXTENDEDMODE
       FILES c018i_e.o
       FILES clib_e.lib
       FILES p18f6722_e.lib
    #else
       FILES c018i.o
       FILES clib.lib
       FILES p18f6722.lib
    #endif
#endif
#endif

#if _DEBUGCODESTART
    CODEPAGE NAME=page START=0x0 END=_CODEEND
    CODEPAGE NAME=debug START=_DEBUGCODESTART END=_CEND
#else
    CODEPAGE NAME=page START=0x0 END=0x1FFFF
#endif

CODEPAGE NAME=idlocs START=0x200000 END=0x200007 PROTECTED
CODEPAGE NAME=config START=0x300000 END=0x30000D PROTECTED
CODEPAGE NAME=devid START=0x3FFFFE END=0x3FFFFF PROTECTED
CODEPAGE NAME=eedata START=0xF00000 END=0xF003FF PROTECTED

#if _EXTENDEDMODE
    DATABANK NAME=gpre START=0x0 END=0x5F
#else
    ACCESSBANK NAME=accessram START=0x0 END=0x5F
#endif

DATABANK NAME=gpr0 START=0x60 END=0xFF
DATABANK NAME=gpr1     START=0x100     END=0x1FF
DATABANK NAME=gpr2     START=0x200     END=0x2FF
DATABANK NAME=gpr3     START=0x300     END=0x3FF
DATABANK NAME=gpr4     START=0x400     END=0x4FF
DATABANK NAME=gpr5     START=0x500     END=0x5FF
DATABANK NAME=gpr6     START=0x600     END=0x6FF
DATABANK NAME=gpr7     START=0x700     END=0x7FF
DATABANK NAME=gpr8     START=0x800     END=0x8FF
DATABANK NAME=gpr9     START=0x900     END=0x9FF
DATABANK NAME=gpr10    START=0xA00     END=0xAFF
DATABANK NAME=gpr11    START=0xB00     END=0xBFF
DATABANK NAME=gpr12    START=0xC00     END=0xCFF
DATABANK NAME=gpr13    START=0xD00     END=0xDFF

#ifdef _DEBUGDATASTART
  DATABANK NAME=gpr14    START=0xE00     END=_DATAEND
  DATABANK NAME=dbgspr   START=_DEBUGDATASTART END=_DEND     PROTECTED
#elif /no debug
  DATABANK NAME=gpr14    START=0xE00     END=0xEFF
#fi

DATABANK NAME=gpr15    START=0xF00     END=0xF5F
ACCESSBANK NAME=accesssfr START=0xF60     END=0xFFF     PROTECTED

#ifdef _CRUNTIME
  SECTION NAME=CONFIG   ROM=config
    #ifdef _DEBUGDATASTART
      STACK SIZE=0x100   RAM=gpr13
    #else
      STACK SIZE=0x100   RAM=gpr14
    #endif
    #fi
#else
    #fi
#fi
Chapter 12. Linker Processing

12.1 INTRODUCTION

Understanding how MPLINK linker processes files and information can be useful to keep in mind when writing and structuring your application code.

Topics covered in this chapter:
- Linker Processing Overview
- Linker Allocation Algorithm
- Relocation Example
- Initialized Data
- Reserved Section Names

12.2 LINKER PROCESSING OVERVIEW

A linker combines multiple input object modules into a single executable output module. The input object modules may contain relocatable or absolute sections of code or data which the linker will allocate into target memory. The target memory architecture is described in a linker command file. This linker command file provides a flexible mechanism for specifying blocks of target memory and for mapping sections to the specified memory blocks. If the linker cannot find a block of target memory in which to allocate a section, an error is generated. The linker combines like-named input sections into a single output section. The linker allocation algorithm is described in Section 12.3 “Linker Allocation Algorithm”.

Once the linker has allocated all sections from all input modules into target memory, it begins the process of symbol relocation. The symbols defined in each input section have addresses dependent upon the beginning of their sections. The linker adjusts the symbol addresses based upon the ultimate location of their allocated sections.

After the linker has relocated the symbols defined in each input section, it resolves external symbols. The linker attempts to match all external symbol references with a corresponding symbol definition. If any external symbol references do not have a corresponding symbol definition, an attempt is made to locate the corresponding symbol definition in the input library files. If the corresponding symbol definition is not found, an error is generated.

If the resolution of external symbols was successful, the linker then proceeds to patch each section's raw data. Each section contains a list of relocation entries which associate locations in a section's raw data with relocatable symbols. The addresses of the relocatable symbols are patched into the raw data. The process of relocating symbols and patching section is described in Section 12.4 “Relocation Example”.

After the linker has processed all relocation entries, it generates the executable output module.
12.3 LINKER ALLOCATION ALGORITHM

The linker allocates memory areas to allow maximum control over the location of code and data, called “sections”, in target memory. There are four kinds of sections that the linker handles:

1. Absolute Assigned
2. Absolute Unassigned
3. Relocatable Assigned
4. Relocatable Unassigned

An absolute section is a section with a fixed (absolute) address that cannot be changed by the linker. A relocatable section is a section that will be placed in memory based on the linker allocation algorithm.

An assigned section is a section that has been assigned a target memory block in the linker command file. An unassigned section is a section that has been left unassigned in this file.

The linker performs allocation of absolute (assigned and unassigned) sections first, relocatable assigned sections next, and relocatable unassigned sections last. The linker also handles stack allocation.

12.3.1 Absolute Allocation

Absolute sections may be assigned to target memory blocks in the linker command file. But, since the absolute section's address is fixed, the linker can only verify that if there is an assigned target memory block for an absolute section, the target memory block has enough space and the absolute section does not overlap other sections. If no target memory block is assigned to an absolute section, the linker tries to find the one for it. If one can not be located, an error is generated. Since absolute sections can only be allocated at a fixed address, assigned and unassigned sections are performed in no particular order.

12.3.2 Relocatable Allocation

Once all absolute sections have been allocated, the linker allocates relocatable assigned sections. For relocatable assigned sections, the linker checks the assigned target memory block to verify that there is space available; otherwise an error is generated. The allocation of relocatable assigned sections occurs in the order in which they were specified in the linker command file.

After all relocatable assigned sections have been allocated, the linker allocates relocatable unassigned sections. The linker starts with the largest relocatable unassigned section and works its way down to the smallest relocatable unassigned section. For each allocation, it chooses the target memory block with the smallest available space that can accommodate the section. By starting with the largest section and choosing the smallest accommodating space, the linker increases the chances of being able to allocate all the relocatable unassigned sections.

12.3.3 Stack Allocation

The stack is not a section but gets allocated along with the sections. The linker command file may or may not assign the stack to a specific target memory block. If the stack is assigned a target memory block, it gets allocated just before the relocatable assigned sections are allocated. If the stack is unassigned, then it gets allocated after the relocatable assigned sections and before the other relocatable unassigned sections are allocated.
12.4 RELOCATION EXAMPLE

The following example illustrates how the linker relocates sections. Suppose the following source code fragment occurred in a file:

/* File: ref.c */
char var1;          /* Line 1 */
void setVar1(void) /* Line 2 */
{
    var1 = 0xFF;      /* Line 3 */
}

Suppose this compiles into the following assembly instructions:

Note: This example deliberately ignores any code generated by MPLAB C18 to handle the function's entry and exit

0x0000 MOVLW 0xFF
0x0001 MOVLB ?? ; Need to patch with var1's bank
0x0002 MOVWF ?? ; Need to patch with var1's offset

When the compiler processes source line 1, it creates a symbol table entry for the identifier var1 which has the following information:

Symbol[index] => name=var1, value=0, section=.data, class=extern

When the compiler processes source line 3, it generates two relocation entries in the code section for the identifier symbol var1 since its final address is unknown until link time. The relocation entries have the following information:

Reloc[index] => address=0x0001 symbol=var1 type=bank
Reloc[index] => address=0x0002 symbol=var1 type=offset

Once the linker has placed every section into target memory, the final addresses are known. Once all identifier symbols have their final addresses assigned, the linker must patch all references to these symbols using the relocation entries. In the example above, the updated symbol might now be at location 0x125:

Symbol[index] => name=var1, value=0x125, section=.data, class=extern

If the code section above were relocated to begin at address 0x50, the updated relocation entries would now begin at location 0x51:

Reloc[index] => address=0x0051 symbol=var1 type=bank
Reloc[index] => address=0x0052 symbol=var1 type=offset

The linker will step through the relocation entries and patch their corresponding sections. The final assembly equivalent output for the above example would be:

0x0050 MOVLW 0xFF
0x0051 MOVLB 0x1 ; Patched with var1's bank
0x0052 MOVWF 0x25 ; Patched with var1's offset
12.5 INITIALIZED DATA

MPLINK linker performs special processing for input sections with initialized data. Initialized data sections contain initial values (initializers) for the variables and constants defined within them. Because the variables and constants within an initialized data section reside in RAM, their data must be stored in nonvolatile program memory (ROM). For each initialized data section, the linker creates a section in program memory. The data is moved by initializing code (supplied with MPLAB C18 and MPASM assembler) to the proper RAM location(s) at start-up.

The names of the initializer sections created by the linker are the same as the initialized data sections with a \_i appended. For example, if an input object module contains an initialized data section named .idata_main.o the linker will create a section in program memory with the name .idata_main.o_i which contains the data.

In addition to creating initializer sections, the linker creates a section named .cinit in program memory. The .cinit section contains a table with entries for each initialized data section. Each entry is a triple which specifies where in program memory the initializer section begins, where in data memory the initialized data section begins, and how many bytes are in the initialized data section. The boot code accesses this table and copies the data from ROM to RAM.

12.6 RESERVED SECTION NAMES

Both the MPASM assembler and the MPLAB C18 C compiler have reserved names for certain types of sections. Please see the documentation for these tools to ensure that you do not use a reserved name for your own section. The linker will be unable to generate the application if there is a section naming conflict.
Chapter 13. Sample Applications

13.1 INTRODUCTION

You can learn the basics of how to use MPLINK linker from the four sample applications listed below. These sample applications can be used as templates for your own application.

- How to Build the Sample Applications
- Sample Application 1 - Templates and Linker Scripts
  - How to find and use template files
  - When to modify the generic linker script file
- Sample Application 2 – Placing Code and Setting Config Bits
  - How to place program code in different memory regions
  - How to place data tables in ROM memory
  - How to set configuration bits in C
- Sample Application 3 – Using a Boot Loader
  - How to partition memory for a boot loader
  - How to compile code that will be loaded into external RAM and executed
- Sample Application 4 – Configuring External Memory
  - How to create new linker script memory section
  - How to declare external memory through #pragma code directive
  - How to access external memories using C pointers

13.2 HOW TO BUILD THE SAMPLE APPLICATIONS

To build the sample applications, you will need the MPASM assembler, the MPLINK linker and, for some sample applications, the MPLAB C compiler for PIC18 MCUs (formerly MPLAB C18) installed on your PC. The assembler and linker are automatically installed with MPLAB IDE, or may be acquired separately on the Microchip website or the C compiler CD-ROM. A free demo (student) version of the C compiler may be obtained on the Microchip website. The full C compiler must be purchased separately.

By default, the tool executables are located as specified in the tables below.

**TABLE 13-1: ASSEMBLY CODE EXECUTABLES AND PATHS**

<table>
<thead>
<tr>
<th>Executables</th>
<th>Default Paths to Executables</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpasmwin.exe</td>
<td>C:\Program Files\Microchip\MPASM Suite</td>
</tr>
<tr>
<td>mplink.exe</td>
<td>C:\Program Files\Microchip\MPASM Suite</td>
</tr>
</tbody>
</table>
13.2.1 Using MPLAB IDE

MPLAB IDE provides a GUI method of developing your code.

13.2.1.1 BUILDING APPLICATIONS

To build an application with MPLAB IDE:

1. Use the Project Wizard under the Project menu to create a project.
   - Select the device specified in the sample application.
   - For assembly applications, select the “Microchip MPASM Toolsuite” as the active toolsuite. For C code applications or combined C code and assembly applications, select the “Microchip C18 Toolsuite” as the active toolsuite. Make sure the executable paths are correct, as per Table 13-1 or Table 13-2, respectively.
   - Name the project and place it in its own folder.
   - Add the sample files to your project, e.g., source1.c, source2.asm and (if customized) script.lkr.

2. Once the project is created, select Project>Build Options>Project to open the Build Options for Project dialog.
   - For MPLAB C compiler for PIC18 MCUs sample applications, click the Directories tab and enter CompilerInstallationPath\bin under “Library Path”, where CompilerInstallationPath is the location where the C compiler is installed on your system.
   - Click the MPLINK Linker tab and then click the “Generate map file” checkbox to select it.

3. Select from the Build Configuration list (see below) whether you will be developing your application (Debug) or are ready to program it into a device (Release).

4. Select Project>Build All to build the application. If your project contains a single assembly file with no linker script file, you will be asked if you want to build “absolute” code or “relocatable” code. The sample applications should be built as relocatable code.

5. If the application fails to build, check that the environment variables discussed in the next section were set correctly during tool installation.

<table>
<thead>
<tr>
<th>Executables</th>
<th>Default Paths to Executables</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcc18.exe</td>
<td>C:\mcc18\bin</td>
</tr>
<tr>
<td>mpasmwin.exe</td>
<td>C:\mcc18\mpasm</td>
</tr>
<tr>
<td>mplink.exe</td>
<td>C:\mcc18\bin</td>
</tr>
</tbody>
</table>

Note: Future C compiler versions may be located at: C:\Program Files\Microchip\MPLAB C18.

Note: Use mplink.exe and not _mplink.exe. The executable file _mplink.exe is not a stand-alone program.
13.2.1.2 EXAMPLE

As an example, consider Sample Application 3, C code mixed boot loader/application.

To build an application with MPLAB IDE:
1. Use the Project Wizard under the Project menu to create a project.
   - Select PIC18F8722 as the device.
   - Select the “Microchip C18 Toolsuite” as the active toolsuite. Make sure the
     executable paths are correct, as per Table 13-2.
   - Name the project and place it in its own folder.
   - Add the sample files to your project, i.e., c018i_mod.c, mixed.asm and
     mixed.lkr.
   - View the Project window (View>Project) to see your project files.

2. Once the project is created, select Project>Build Options>Project to open the
Build Options for Project dialog.
   - Click the Directories tab and enter C:\mcc18\lib under "Library Path".
   - Click the MPLINK Linker tab and then click the “Generate map file” checkbox
     to select it.
3. Select “Debug” from the Build Configuration list on the Project Manager toolbar.
4. Select Project>Build All to build the application.
13.2.2 Using the Command Line

The command line provides a platform independent method to develop your code.

13.2.2.1 BUILDING APPLICATIONS

To build an application on the command line:

1. The listed Environment Variables need to be set as specified. To set these variables, go to the Command prompt and type SET to view and set the variables. In Windows OS, go to Start>Settings>Control Panel>System, Advanced tab, Environment Variables button. View and edit variables here.
   a) PATH - Make sure the executables can be found, as per Table 13-1 and Table 13-2.
   b) MCC_INCLUDE - If MPLAB C Compiler for PIC18 MCUs is used, this should point to the \h subdirectory of the C compiler installation directory.

2. For C code compilation, use the following:
   ```
   mcc18 -p device source1.c
   ```
   where `device` is the device representation (e.g., 18F8772 for the PIC18F8722 device) and `source1.c` is the C code source file example. For multiple files, leave a space between each file.

3. For MPASM assembly, use the following:
   ```
   mpasmwin -p device source2.asm
   ```
   where `device` is the device representation and `source2.asm` is the assembly code source file example. For multiple files, leave a space between each file.

4. For MPLINK linking of files to create an application, use (shown on two lines but type on one):
   ```
   mplink /p device /u_macro source1.o source2.o
   /l c:\mcc18\lib /m app.map
   ```
   or (for a modified linker script):
   ```
   mplink modified.lkr /u_macro source1.o source2.o
   /l c:\mcc18\lib /m app.map
   ```
   where:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>device</code></td>
<td>The device representation.</td>
</tr>
<tr>
<td><code>modified.lkr</code></td>
<td>The modified linker script file.</td>
</tr>
<tr>
<td><code>_macro</code></td>
<td>A conditional macro (e.g., <code>_CRUNTIME</code>). For more details, see Section 11.8 “Conditional Linker Statements”.</td>
</tr>
<tr>
<td><code>source1.o</code></td>
<td>A C code object file.</td>
</tr>
<tr>
<td><code>source2.o</code></td>
<td>An assembly code object file.</td>
</tr>
<tr>
<td><code>c:\mcc18\lib</code></td>
<td>The library path, only needed if the MPLAB® C compiler for PIC18 MCUs was used, as here to generate <code>source1.o</code> from <code>source1.c</code>.</td>
</tr>
<tr>
<td><code>app.map</code></td>
<td>The map file.</td>
</tr>
</tbody>
</table>
13.2.2.2 EXAMPLE

As an example, consider Sample Application 3, C code mixed boot loader/application.

To build an application on the command line:

1. The listed Environment Variables need to be set as specified.
   a) PATH - Make sure the executables can be found, as per Table 13-2.
   b) MCC_INCLUDE - Point to the c:\mcc18\h subdirectory.
2. For C code compilation, use the following:

   mcc18 -p 18F8722 c018i_mod.c mixed.c

3. For MPLINK linking of files to create an application, use (shown on two lines but type on one):

   mplink mixed.lkr /u_CRUNTIME c018i_mod.o mixed.o
   /l c:\mcc18\lib /m mixed.map

   If you want to link for debug mode or PIC18 extended mode, see Section 11.8 “Conditional Linker Statements”.

© 2009 Microchip Technology Inc.

DS33014K-page 199
13.3 SAMPLE APPLICATION 1 - TEMPLATES AND LINKER SCRIPTS

In the MPLAB IDE installation, assembly source code templates and generic linker script files are provided for most devices supported by MPLAB IDE. The source code templates give you a starting point from which to begin coding. The generic linker scripts are used automatically by the linker to simplify application development. Or you may modify a linker script and then manually add it on the linker command line or to the project.

This first general example will discuss templates and linker scripts.

13.3.1 Locating Templates and Linker Script Files

For MPLAB IDE installed in the default location, source code templates may be found at:

C:\Program Files\Microchip\MPASM Suite\Template

in the following subdirectories:

- Code - Contains absolute assembly code examples by device
- Object - Contains relocatable assembly code examples by device

The relocatable source code template 18F8722TMO.ASM for the PIC18F8722 may be found in the Object directory. This template provides oscillator setup, example variable and EEPROM setup, reset and interrupt handling routines, and a section main for application code.

For MPLAB IDE installed in the default location, the generic linker script files may be found at:

C:\Program Files\Microchip\MPASM Suite\LKR

The generic linker script file for the PIC18F8722 would be 18f8722_g.lkr. This file defines initialization files, program code sections, GPR sections, and access memory sections. These definitions are grouped based on debugger or programmer usage, and regular or extended memory usage.

Program memory sections specified by the template code and linker script are compared below.

**TABLE 13-3: PROGRAM MEMORY MAP – PIC18F8722**

<table>
<thead>
<tr>
<th>Program Memory Address</th>
<th>Linker Script Section (Not debug or extended)</th>
<th>Template Source Code Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 0x0007</td>
<td>page - ROM code space</td>
<td>RESET_VECTOR - Reset vector</td>
</tr>
<tr>
<td>0x0008 0x0017</td>
<td></td>
<td>HI_INT_VECTOR - High priority interrupt vector</td>
</tr>
<tr>
<td>0x0018</td>
<td></td>
<td>LOW_INT_VECTOR - Low priority interrupt vector</td>
</tr>
<tr>
<td>0x0019 0x01FFFF</td>
<td></td>
<td>High_Int - High priority interrupt handler Low_Int - Low priority interrupt handler Main - Main application code</td>
</tr>
<tr>
<td>0x200000 0x200007</td>
<td>idlocs - ID locations</td>
<td></td>
</tr>
<tr>
<td>0x300000 0x3000D</td>
<td>config - Configuration bits</td>
<td>CONFIG - Configuration settings</td>
</tr>
<tr>
<td>0x3FFFFFF 0x3FFFFFF</td>
<td>devid - Device ID</td>
<td></td>
</tr>
<tr>
<td>0xFO0000 0xF003FF</td>
<td>eedata - EEPROM data</td>
<td>DATA_EEPROM - Data EEPROM</td>
</tr>
</tbody>
</table>
13.3.2 Modifying Templates and Linker Script Files

For this sample application, assembly code is added to the template but the generic linker script is not edited.

In general, you will modify templates to create your own application code but you should not need to modify the generic linker script file. Still, there are reasons why you might want to customize a linker script, as shown in the other sample applications.

In addition, a case where you might want to modify the linker script is when you want to use a C code data object that is larger than 256 bytes, such as a large array. This application is discussed in the *MPLAB C Compiler for PIC18 MCUs Getting Started* (DS51295), FAQ-10.

**Modified 18F8722TMPO.ASM**

In the template, add the following udata section:

```asm
; Oscillator Selection:
CONFIG OSC = LP ;LP

; Array variables
array UDATA 0x2FE
element1 RES 1
element2 RES 1
element3 RES 1
element4 RES 1
element5 RES 2

; Variable definitions
```

Then, in the main code portion, add the following program code:

```asm
; Start of main program
; The main program code is placed here.
Main:

; *** main code goes here ***
banksel element1 ;Select element1 bank
movlw 0x55 ;Move literal
movwf element1,1 ;to element1
movff element1,element2 ;Move element1
rrcf element2,1,1 ;to element2, right shift
movff element2,WREG ;element2, move to WREG
; Since regular RAM was
; used instead of Access
; RAM, change physical banks
banksel element3 ;Select element3 bank
movwf element3 ;Move WREG to element3,
rrcf element3,1,1 ;right shift and
movff element3,element4 ;move to element4
rrcf element4,1,1 ;Right shift element4
movff element4,element5 ;and move to element5
;low byte
rrcf element4,1,1 ;Right shift element4
movff element4,element5+1 ;again and move to
element5 high byte
```
13.3.3   Building the Application

To build the application, see Section 13.2 “How to Build the Sample Applications”. Then, to continue development with MPLAB IDE, you could place the element variables in a Watch window to see their values, remembering to change \texttt{element5} to a 16-bit value (Properties dialog, Watch Properties tab).
13.4 SAMPLE APPLICATION 2 – PLACING CODE AND SETTING CONFIG BITS

This example is for the PIC18F8720 in extended microcontroller mode.

The file `eeprom2.asm` places interrupt handling code at 0x20000 (external memory.) The assembly code directive, `INTHAND CODE`, places the code that follows into the `INTHAND` section. The modified linker script file (`eeprom.lkr`) maps the `INTHAND` section to the `CODE` region that begins at 0x20000.

The file `eeprom1.c` has a 0x1000 element data table in program memory in the same code page with the interrupt handlers. The data table is defined in C using the `#pragma romdata` directive to place the table in a section called `DATTBL`. The modified linker script file maps the `DATTBL` section to the `CODE` region that begins at 0x20000.

Additionally, configuration bits are set in C using the `#pragma config` directive.

The main function in the C file is placed in the default `CODE` section because there is no section directive explicitly assigned.

For additional information, you may wish to reference:

- PIC18F8720 Device Data Sheet (DS39609)
- MPLAB C18 C Compiler User’s Guide (DS51288)
- External Memory Interfacing Techniques for the PIC18F8XXX (AN869)

Program memory sections specified by the code and linker script are compared below. Specific sections highlighted in this sample application (SA2) are noted.

**TABLE 13-4: PROGRAM MEMORY MAP - PIC18F8720**

<table>
<thead>
<tr>
<th>SA2</th>
<th>Program Memory Address</th>
<th>Linker Script Section (Not debug or extended)</th>
<th>Source Code Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x000000</td>
<td>page - On-chip Memory</td>
<td>STARTUP</td>
</tr>
<tr>
<td></td>
<td>0x01FFFF</td>
<td></td>
<td>PROG - Main Application Code</td>
</tr>
<tr>
<td>→</td>
<td>0x020000</td>
<td>eeprom - External Memory</td>
<td>INTHAND</td>
</tr>
<tr>
<td></td>
<td>0x1FFFFF</td>
<td></td>
<td>- Interrupt Handler</td>
</tr>
<tr>
<td></td>
<td>0x200000</td>
<td>idlocs - ID Locations</td>
<td>DATTBL</td>
</tr>
<tr>
<td></td>
<td>0x200007</td>
<td></td>
<td>- Data Table</td>
</tr>
<tr>
<td>→</td>
<td>0x300000</td>
<td>config - Configuration Bits</td>
<td>CONFIG</td>
</tr>
<tr>
<td></td>
<td>0x30000D</td>
<td></td>
<td>- Configuration Settings</td>
</tr>
<tr>
<td></td>
<td>0x3FFFFE</td>
<td>devid - Device ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x3FFFFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xF00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xF00007</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xF003FF</td>
<td>eedata - EE Data</td>
<td></td>
</tr>
</tbody>
</table>
13.4.1  C Source Code - eeprom1.c

/* eeprom1.c */
#include <p18f8720.h>
#define DATA_SIZE 0x1000

/* Data Table Setup */
#pragma romdata DATTBL // Put following romdata into section DATTBL
unsigned rom data[DATA_SIZE];
#pragma romdata        // Set back to default romdata section

/* Configuration Bits Setup */
The #pragma config directive specifies the processor-specific configuration settings (i.e., configuration bits) to be used by the application. For more on this directive, see the "MPLAB C18 C Compiler User's Guide" (DS51288). */
#pragma config OSCS = ON, OSC = LP // Enable OSC switching and LP
#pragma config PWRT = ON           // Enable POR
#pragma config BOR = ON, BORV = 42 // Enable BOR at 4.2v
#pragma config WDT = OFF           // Disable WDT
#pragma config MODE = EM           // Use Extended MCU mode

/* Main application code for default CODE section */

void main( void )
{
    while( 1 )
    {
        } // end while
    } // end main

13.4.2  Assembler Source Code - eeprom2.asm

; eeprom2.asm

list p=18f8720

#include p18f8720.inc

INTHAND code

; place interrupt handling code in here

end
13.4.3 Linker Script - eeprom.lkr

The linker script file eeprom.lkr is a modified version of the generic linker script file 18f8720_g.lkr. Modify the generic linker script as follows to create eeprom.lkr.

Add EEPROM codepage information:

```c
#ifdef _DEBUGCODESTART
   CODEPAGE  NAME=page      START=0x0          END=_CODEEND
   CODEPAGE  NAME=debug     START=_DEBUGCODESTART END=_CEND        PROTECTED
#endif

//EEPROM codepage
CODEPAGE  NAME=eeprom     START=0x20000       END=0x1FFFFF       PROTECTED
CODEPAGE  NAME=idlocs     START=0x200000      END=0x200007       PROTECTED
CODEPAGE  NAME=config     START=0x300000      END=0x30000D       PROTECTED
CODEPAGE  NAME=devid      START=0x3FFFFE      END=0x3FFFFF       PROTECTED
CODEPAGE  NAME=eedata     START=0xF00000      END=0xF003FF       PROTECTED

#ifdef _CRUNTIME
   SECTION   NAME=CONFIG   ROM=config
   SECTION   NAME=INTHAND  ROM=eeprom     // Interrupt handlers
   SECTION   NAME=DATTRL   ROM=eeprom     // Data tables
#endif

#ifdef _DEBUGDATASTART
   STACK SIZE=0x100 RAM=gpr13
#else
   STACK SIZE=0x100 RAM=gpr14
#endif
#endif
```

Then add section information for the EEPROM sections:

```c
#ifdef _DEBUGCODESTART
   SECTION   NAME=page      ROM=config
   SECTION   NAME=debug     ROM=eeprom     // Interrupt handlers
#endif
```

13.4.4 Building the Application

To build the application, see Section 13.2 “How to Build the Sample Applications”. Then, to continue development with MPLAB IDE:

1. Though the configuration bits in code set the microcontroller mode to external, you must tell MPLAB IDE the range of external memory you wish to use. Select Configure>External Memory. In the dialog, click “Use External Memory” and enter “0x1FFFFF” as the "Address Range End". Click OK.

2. Select Project>Build All to build the application again.

13.4.5 Absolute Method

Instead of adding SECTION lines to the linker script, you could add the absolute address in code, i.e.,

```
INTHAND code 0x20000
```

However, if you need to place additional code in the CODEPAGE eeprom area, you would need to know the disassembly code length of the interrupt handler to determine the absolute address at which to place this additional code. Also, if you edit the interrupt handler code, you would need to remember to change the address of the additional code.

Therefore, it is usually easier not to place code absolutely but to allow the linker script to place code for you.
13.5 SAMPLE APPLICATION 3 – USING A BOOT LOADER

A boot loader is a special program that, when programmed into the target PIC microcontroller, is responsible for downloading and programming relocatable application code into the same target PIC microcontroller. The relocatable application or “user” code is typically transferred to the boot loader through serial communications, such as RS232.

13.5.1 C Compiler Usage

This section discusses how to use the MPLAB C Compiler for PIC18 MCUs (the C compiler) when developing bootloader and related application code.

There are three examples showing how to modify the C compiler linker scripts and how to use the #pragma code directive in the source code for the C compiler boot loader project. To better understand how the code corresponds to locations in device program memory, see 13.5.1.1 “C Compiler Memory Map”.

Example 1 shows how to configure the C compiler linker script and suggests how to use code directives for the C compiler boot loader. See 13.5.1.2 “Example 1: C Compiler Boot Loader”.

Example 2 shows the C compiler linker script configuration and suggested code directives for the C compiler application targeted for a microcontroller that is running the C compiler boot loader. See 13.5.1.3 “Example 2: C Compiler Application”.

Example 3 is a mixed language example using the C compiler application targeted for a microcontroller, such as the PIC18F8720 with a limited boot block size, running an MPASM boot loader. A boot loader written in C code will typically require more program memory than a boot loader written in assembly and therefore requires a microcontroller with a larger boot block region, such as the PIC18F8722. See 13.5.1.4 “Example 3: Mixed Language Boot Loader/Application”.

Boot loader and application code written for the C compiler must use the C compiler linker scripts to command the linker to place the compiled C source code into appropriate program memory sections. Typically, boot loader code is compiled and linked for a destination in the “boot” section of the target microcontroller’s program memory. The “application” code is compiled and linked for a destination inside the user section of program memory.

To build the C compiler sample application, refer to Section 13.2 “How to Build the Sample Applications”.

13.5.1.1 C COMPILER MEMORY MAP

The first two C compiler boot loader examples are demonstrated using a PIC18F8722 which offers a configurable boot block size of 2K, 4K or 8K bytes. The remaining program memory is available for the relocatable application code and data tables. For these two examples it is assumed the boot block is configured for 2K bytes and requires modification to the C compiler linker script file in order to accommodate the selected boot block size.

The third example, a mix of an MPASM assembler boot loader and C compiler source code, uses the PIC18F8720. For the corresponding memory map, see Section 13.5.2.1 “Assembler Memory Map”.

For the first two examples, program memory sections specified by the code and linker script are compared below. Specific sections highlighted in these sample application (SA3) examples are noted.

<table>
<thead>
<tr>
<th>TABLE 13-5: PROGRAM MEMORY MAP - PIC18F8722</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SA3</strong> Program Memory Address</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td>0x0000000 0x000029</td>
</tr>
<tr>
<td>0x00002A 0x0007FF</td>
</tr>
<tr>
<td>0x000800 0x1FFFFF</td>
</tr>
</tbody>
</table>

13.5.1.2 EXAMPLE 1: C COMPILER BOOT LOADER

This example shows a section of linker script modified to accommodate the boot loader code.

**Boot Loader Linker Script**

The partial C compiler linker script file shown below demonstrates the modifications required to the generic linker script when building the C compiler boot loader source code files. The MPLINK linker will use this configuration to link the compiled source code into the boot program memory region starting at 002Ah. The vector locations will be specified in the boot loader source code using the appropriate #pragma code directives.

```
CODEPAGE NAME=vectors START=0x0 END=0x29
CODEPAGE NAME=boot START=0x2A END=0x7FF

#ifdef _DEBUGCODESTART
CODEPAGE NAME=page START=0x800 END=_CODEEND
CODEPAGE NAME=debug START=_DEBUGCODESTART END=_CEND PROTECTED
#else
CODEPAGE NAME=page START=0x800 END=0x1FFFF
#endif

CODEPAGE NAME=idlocs START=0x200000 END=0x200007 PROTECTED
CODEPAGE NAME=confi START=0x300000 END=0x30000D PROTECTED
CODEPAGE NAME=devid START=0x3FFFFF END=0x3FFFFF PROTECTED
CODEPAGE NAME=eedata START=0xF00000 END=0xF003FF
```
Boot Loader Source Code

The C compiler boot loader code can be composed of one or more aggregate relocatable C source files that are compiled and linked together during build time. In this example, the source code file uses the #pragma code directive to instruct the linker to place the interrupt vectors at memory locations 0008h and 0018h. A "main" function must be included, as this is called from the C compiler startup code that is added during link process.

```c
#include <p18cxxx.h>
#define RM_RESET_VECTOR 0x000800 // define relocated vector addresses
#define RM_HIGH_INTERRUPT_VECTOR 0x000008
#define RM_LOW_INTERRUPT_VECTOR 0x000018

/** VECTOR MAPPING ******************************************/
#pragma code _HIGH_INTERRUPT_VECTOR = 0x000008
void _high_ISR (void)
{
    _asm goto RM_HIGH_INTERRUPT_VECTOR _endasm
}

#pragma code _LOW_INTERRUPT_VECTOR = 0x000018
void _low_ISR (void)
{
    _asm goto RM_LOW_INTERRUPT_VECTOR _endasm
}
/** BOOT LOADER CODE ******************************************/
#pragma code
void main(void)
{
    //Check Bootload Mode Entry Condition
    if(PORTBbits.RB4 == 1) // If not pressed, User Mode
    {
        _asm goto RM_RESET_VECTOR _endasm
    }
    //Else continue with bootloader code here ...
}
#pragma code user = RM_RESET_VECTOR  // This address defined as 0x800 above
                        // or can be defined in header file
/** END OF BOOT LOADER ******************************************/
```

13.5.1.3 EXAMPLE 2: C COMPILER APPLICATION

This example shows a section of linker script modified to accommodate the application code.

Application Linker Script

The boot loader linker script file may be used when building the C compiler application source code files. The linker will use this configuration to link the compiled source code into the page1 program memory region above the protected boot loader region.
Application Source Code

The C compiler application code can be composed of one or more aggregate relocatable C source files that are compiled and linked together during build time. In the code snippet shown below, the source code file uses the #pragma code directive to instruct the linker to place the relocated reset and interrupt vectors at the appropriate memory locations. A main function must be included, as this is called from the C compiler startup code that is added during the link process. The linker automatically includes this C compiler initialization code provided in file c018i.c and must be accessed by the application code through an “in-line” assembly goto instruction shown below.

```
#include <p18cxxx.h>

/** VECTOR MAPPING ************************************************/
extern void _startup (void);   // See c018i.c in your C18 compiler dir

#pragma code _RESET_INTERRUPT_VECTOR = 0x000800
void _reset (void)
{    
    _asm goto _startup _endasm
 }

#pragma code _HIGH_INTERRUPT_VECTOR = 0x000808
void _high_ISR (void)
{    
    ;
 }

#pragma code _LOW_INTERRUPT_VECTOR = 0x000818
void _low_ISR (void)
{    
    ;
 }

/** APPLICATION CODE*******************************************/
#pragma code
void main(void)
{    
    while(1)
    {    
        ;   // Main application code here
    }
}

/** END OF APPLICATION ******************************************/
```
13.5.1.4 EXAMPLE 3: MIXED LANGUAGE BOOT LOADER/APPLICATION

This example shows the linker script, startup code, and source code modifications needed to accommodate a mixed language application which employs a boot loader.

**Mixed Language Linker Script**

The partial C compiler linker script file shown below demonstrates the required modifications when building the mixed assembly boot loader/C code application. The linker will use this configuration to link the compiled source code into the user program memory region above the protected boot loader. In this linker script example, the C compiler start-up file `c018i.o` has been remarked out, preventing the linker from linking this object file to the project. Instead it will use the modified file in the next section.

```
#IFDEF _CRUNTIME
#IFDEF _EXTENDEMODE
FILES c018i_e.o
FILES clib_e.lib
FILES p18f8722_e.lib

#else
    FILES c018i.o     <-- Note this line to be ignored by linker
FILES clib.lib
FILES p18f8722.lib
#FI
#FI
```

```
CODEPAGE   NAME=vectors    START=0x0               END=0x29
CODEPAGE   NAME=boot       START=0x2A              END=0x1FF

#if _DEBUGCODESTART
CODEPAGE   NAME=page       START=0x200             END=_CODEEND  PROTECTED
#endif
```

**Mixed Language c018i.c Modifications**

For a typical C compiler application, the `c018i.c` startup code normally specifies program memory location 0000h as the entry section and is linked into the project by the linker when specified in the MPLAB C18 linker script. Since the C compiler application code in this example has been relocated to program memory address 0200h because of the boot loader, it is necessary to change the code section `_entry_scn` definition in `c018i.c` file as shown below and to add the `c018i.c` source file to the project for recompiling and linking.

```
#pragma code _entry_scn=0x000200
void
_entry (void)
{
    _asm goto _startup _endasm
}
```
**Mixed Language Source Code**

The C compiler application code can be composed of one or more relocatable C source files that are compiled and linked together during build time. In the code snippet shown below, the source code file uses the `#pragma code` directive to instruct the linker to place the relocated reset and interrupt vectors at the appropriate memory locations. A `main` function must be included, as this is called from the C compiler startup code that is added during the link process.

```c
#include <pl8cxxx.h>

/** VECTOR MAPPING **************************************************************************/

#pragma code _HIGH_INTERRUPT_VECTOR = 0x000208
void _high_ISR (void)
{
    ;            // ISR goes here
}

#pragma code _LOW_INTERRUPT_VECTOR = 0x000218
void _low_ISR (void)
{
    ;            // ISR goes here
}

/** APPLICATION CODE*************************************************************************/
#pragma code
void main(void)
{
    while(1)
    {
        ;            // Main application code here
    }
}

/** END OF APPLICATION **************************************************************************/

13.5.2 Assembler Usage

This section discusses how to use the MPASM assembler (the assembler) when developing bootloader and related application code.

There are three assembler examples showing suggested linker script modifications and appropriate source code directive usage for a boot loader and application project. To better understand how the code corresponds to locations in device program memory, see Section 13.5.2.1 “Assembler Memory Map”.

The modified linker script file provided in this example is designed to support all three of the following examples. See Section 13.5.2.2 “Assembler Linker Script”.

Example 1 shows an assembler boot loader. See Section 13.5.2.3 “Example 1: Assembler Boot Loader Source Code”.

Example 2 shows a multiple module relocatable assembler application. See Section 13.5.2.4 “Example 2: Assembler Application Source Code”.

Example 3 incorporates both the assembler boot loader and multiple module relocatable assembler application as a single program memory image. See Section 13.5.2.5 “Example 3: Assembler Boot Loader/Application Source Code”.

To build the assembler sample application, refer to Section 13.2 “How to Build the Sample Applications”.
13.5.2.1 ASSEMBLER MEMORY MAP

The boot loader typically resides in the “boot block” region of the PIC18F8720’s program memory, which is the first 512 bytes of memory, from location 0000h to 01FFh. The remaining program memory, starting at location 0200h, is available for relocatable application code and data lookup tables. Other PIC18F microcontrollers offer larger boot block regions and will require slightly different linker script modifications than what is represented in this example. However, the concepts shown here can be migrated to these other PIC microcontrollers.

For this example, program memory sections specified by the code and linker script are compared below. Specific sections highlighted in this sample application (SA3) example are noted.

**TABLE 13-6: PROGRAM MEMORY MAP - PIC18F8720**

<table>
<thead>
<tr>
<th>SA3</th>
<th>Program Memory Address</th>
<th>Linker Script Section</th>
<th>Source Code Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x000000</td>
<td>vectors - Reset, Interrupts</td>
<td>Vectors, IntH, IntL</td>
</tr>
<tr>
<td></td>
<td>0x000029</td>
<td>boot_code - Boot Loader</td>
<td>Boot</td>
</tr>
<tr>
<td></td>
<td>0x000200</td>
<td>page - Remapped Vectors, User Code, Data Tables</td>
<td>R_vectors, R_IntH, R_IntL, user_code</td>
</tr>
</tbody>
</table>

13.5.2.2 ASSEMBLER LINKER SCRIPT

To protect the boot block and vector memory regions, the linker script file uses modified CODEPAGE directives to establish these memory regions and uses the PROTECTED modifier to prevent the linker from assigning any relocatable code that is not explicitly assigned to these regions.

The section of modified generic linker script below shows how the linker can assign the relocatable application code to the user code memory region (page) that is not protected. The other program memory regions can only be populated if the CODE directive used in the source files specifies placement of code within these protected memory regions. This linker script file is designed to accommodate all three boot loader design considerations demonstrated in this chapter.

**boot.lkr** - The linker script file for boot loader and application code example projects.

| CODEPAGE NAME=vectors START=0x0 END=0x29 | PROTECTED |
| CODEPAGE NAME=boot START=0x2A END=0x1FF | PROTECTED |
| #IFDEF _DEBUGCODESTART CODEPAGE NAME=page START=0x200 END=_CODEEND |
| CODEPAGE NAME=debug START=_DEBUGCODESTART END=_CEND | PROTECTED |
| #ELSE CODEPAGE NAME=page START=0x200 END=0x1FFFF |
| #FI |
| CODEPAGE NAME=idlocs START=0x200000 END=0x200007 | PROTECTED |
| CODEPAGE NAME=config START=0x300000 END=0x30000D | PROTECTED |
| CODEPAGE NAME=devid START=0x3FFFFF END=0x3FFFFF | PROTECTED |
| CODEPAGE NAME=eedata START=0xF00000 END=0xF003FF | PROTECTED |
13.5.2.3 EXAMPLE 1: ASSEMBLER BOOT LOADER SOURCE CODE

In this example, the boot loader is a single source file that will not be linked with any other source code at build time. The \texttt{CODE} directives used in the boot loader source code instructs the linker to place the reset and interrupt vectors at their appropriate program memory locations for the PIC microcontroller and to place the starting location of the boot loader executable code just above this region starting at location 002Ah.

The program memory section names \texttt{Vectors}, \texttt{IntH} and \texttt{IntL} are used with the \texttt{CODE} directive to instruct the linker to place the assembled code that follows each directive at the specified program memory location. In this case, the boot loader is not linked with any application code so the relocated reset and interrupt vectors, 0208h, 0218h and 022Ah are assumed and therefore are explicitly coded.

\texttt{18Fboot.asm} - This is an example of how the startup portion of a boot loader could be configured when designing and programming only the boot loader code into the target PIC microcontroller.

```assembly
; ****************************************************************************
; 18Fboot.asm
; ****************************************************************************
LIST P=18F8720
#include P18cxxx.inc
; ****************************************************************************
Vectors   code  0x0000
VReset:   bra   Boot_Start

IntH      code  0x0008
VIntH:    bra   0x0208       ; Re-map Interrupt vector to app's code space

IntL      code  0x0018
VIntL:    bra   0x0218       ; Re-map Interrupt vector to app's code space

; ****************************************************************************
Boot      code   0x002A      ; Boot loader executable code starts here
Boot_Start:

; Logic to determine if bootloader executes or branch to user's code
; ...
bra   0x022A       ; Branch to user's application code
; ...
; end of boot loader code section
; ****************************************************************************
END
```

© 2009 Microchip Technology Inc.
13.5.2.4 EXAMPLE 2: ASSEMBLER APPLICATION SOURCE CODE

In this example the application code is composed of several relocatable source files that are assembled and linked together during build time. The relocatable reset and interrupt vector locations are defined in main.asm and are assigned to a specific program memory location by the CODE directive.

**main.asm** - This is a sample of the startup portion of a main source code file that contains the relocated reset and interrupts and is the main entry point into the application.

```assembly
; *****************************************************************************
; main.asm
; *****************************************************************************
LIST P=18F8720
#include P18cxxx.inc
; *****************************************************************************
R_vectors code 0x200
RVReset:                     ;Re-mapped RESET vector
   bra main
R_IntH code 0x208      ;Re-mapped HI-priority interrupt vector
RVIntH:        ;High priority interrupt vector code here
   ;...
   retfie
R_IntL code 0x218      ;Re-mapped LOW-priority interrupt vector
RVIntL:        ;Low priority interrupt vector code here
   ;...
   retfie
user_code code 0x22A
main:          ; Entry into application code starts here
   ;....
; end of main code section
; *****************************************************************************
END
```

```
13.5.2.5  EXAMPLE 3: ASSEMBLER BOOT LOADER/APPLICATION SOURCE CODE

The final example demonstrates the possibility of combining both the boot loader and application code into a single program memory image that can be programmed into a target microcontroller at the same time. Since the boot loader will be assembled and linked with the application source code files, any references to external labels, defined in the application code, must be resolved by the linker. To accomplish this, the GLOBAL directive used in main.asm and the EXTERN directive used in the boot loader source file allow the linker to resolve the relocated reset and interrupt vector labels defined in main.asm and referenced in the 18Fboot_r.asm. For this example, the same boot.lkr linker script file used in the previous examples is used to link the boot loader and application files together.

**18Fboot_r.asm** - This sample version of the boot loader allows for relocatable vectors that are defined, not in the boot loader, but in the application source code.

```assembly
; **************************************************************************
; 18Fboot_r.asm
; **************************************************************************
LIST P=18F8720
#include P18cxxx.inc

; Declare labels used here but defined outside this module
extern RVReset, RVIntH, RVIntL

; Vectors    code  0x0000
VReset:     bra   Boot_Start

IntH        code  0x0008
VIntH:      bra   RVIntH       ; Re-map Interrupt vector

IntL        code  0x0018
VIntL:      bra   RVIntL       ; Re-map Interrupt vector

; Boot        code   0x002A   ; Define explicit Bootloader location
Boot_Start:

; Determine if bootloader should execute or branch to user's code
; ....
bra   RVReset       ; Branch to user's application code
; Else Bootloader execution starts here
; ....

; **************************************************************************
END
```
main_r.asm - This is a sample version of a main source code file that uses the GLOBAL directive to make the relocatable reset and interrupt vector labels available to the boot loader.

```asm
;******************************************************************************
; main_r.asm
;******************************************************************************
LIST P=18F8720
#include P18cxxx.inc

; Define labels here but used outside this module
  global   RVReset, RVIntH, RVIntL
;******************************************************************************
R_vectors   code   0x200
RVReset:                      ;Re-mapped RESET vector
  bra main

R_IntH      code   0x208      ;Re-mapped HI-priority interrupt vector
RVIntH:
;High priority interrupt vector code here
  ;...
  retfie

R_IntL      code   0x218      ;Re-mapped LOW-priority interrupt vector
RVIntL:
;Low priority interrupt vector code here
  ;...
  retfie

user_code   code   0x22A
main:
; Entry into application code starts here
  ; ....
; end of main code section
;******************************************************************************
END
```
13.6 SAMPLE APPLICATION 4 – CONFIGURING EXTERNAL MEMORY

Most of the larger pin count PIC microcontrollers have the ability to interface to external 8- or 16-bit data FLASH or SRAM memories through the External Memory Bus (EMB). The PIC18F8722, for example, has 128K bytes of internal program memory (00000h - 1FFFFFh). But, when configured for Extended Microcontroller mode, external program memory space from locations 20000h through 1FFFFFh becomes externally addressable through the EMB created from the I/O pins.

The use of a linker script file can be extended to other external memory-mapped devices such as programmable I/O peripherals, real-time clocks or any device that has multiple configuration or control registers that can be accessed through an 8- or 16-bit data bus.

13.6.1 C Compiler Usage

This section discusses how to use the MPLAB C Compiler for PIC18 MCUs (the C compiler) when developing external memory application code.

The C compiler linker script file for the PIC18F8722 is modified to instruct the linker that a new memory region is available by adding a CODEPAGE definition as shown below.

```
CODEPAGE NAME=xrasm START=0x020000 END=0x01FFFFF PROTECTED
```

In addition to the new CODEPAGE, a new logical SECTION is created and assigned to the program memory region specified in the associated CODEPAGE definition.

```
SECTION NAME=SRAM_BASE ROM=xrasm
```

In the C compiler application's source code file, the #pragma romdata directive instructs the linker to allocate the SRAM's starting address to the memory region specified by the SRAM_BASE logical section definition. The physical address is provided by the xrasm codepage directive at 20000h. Since the memory region occupied by the SRAM is program memory, not data memory, the rom qualifier is required in the declaration of the char array variable, sram[]. In addition, this memory region is beyond a 16-bit address range (64Kbyte) and therefore requires the use of the far qualifier in order for C pointers to correctly access this region.

```
#pragma romdata SRAM_BASE ;Assigns this romdata space at 0x020000
rom far char sram[];     ;Declare an array at starting address
```

To build the C compiler sample application, refer to Section 13.2 “How to Build the Sample Applications”.

The large memory model must be used in this project.

- For MPLAB IDE, at the end of Step 2, click the MPLAB C18 tab and chose the Category of “Memory Model” from the drop-down list. Under “Code Model”, click “Large code mode (>64K)”.
- For the command line, use the -ml option when compiling.

13.6.1.1 C COMPILER MEMORY MAP

The table below shows the memory mapping for the PIC18F8722 when used with the 1Mbyte external SRAM device. Notice that the first 128K bytes of the external memory region is overlapped with the 128K bytes of internal program memory space and therefore cannot be accessed using the external memory bus. Without any additional external memory address decoding, the first 128K bytes of the SRAM are not accessible and therefore the first addressable location of SRAM is 20000h, as used in this example (SA4).
TABLE 13-7: PROGRAM MEMORY MAP – PIC18F8722 AND 1 MB SRAM

<table>
<thead>
<tr>
<th>SA4</th>
<th>Program Memory Address</th>
<th>SRAM Address</th>
<th>Linker Script Section</th>
<th>Source Code Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x000000 0x01FFFF</td>
<td>0x000000 0x01FFFF</td>
<td>page - Reset, Interrupt vectors and On-chip Memory</td>
<td></td>
</tr>
<tr>
<td>→</td>
<td>0x020000 0x0FFFFF</td>
<td>0x020000 0x0FFFFF</td>
<td>xsram - External Memory</td>
<td>SRAM_BASE - romdata space</td>
</tr>
<tr>
<td></td>
<td>0x100000 0x1FFFFF</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

13.6.1.2 C COMPILER LINKER SCRIPT

The sections of modified PIC18F8722 C compiler generic linker script file shown below demonstrates suggested modifications for external memory applications.

First the CODEPAGE statement is added.

```
CODEPAGE   NAME=xrsm        START=0x020000           END=0x1FFFFF       PROTECTED
CODEPAGE   NAME=idlocs     START=0x200000           END=0x200007       PROTECTED
CODEPAGE   NAME=config     START=0x300000           END=0x30000D       PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE           END=0x3FFFFF       PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000           END=0xF003FF       PROTECTED
```

Then the external memory section is added.

```
#if defined _CRUNTIME
SECTION    NAME=CONFIG     ROM=config
SECTION    NAME=SRAM_BASE  ROM=xrsm
#elif defined _DEBUGDATASTART
// Stack size=0x100 RAM=gpr13
#else
// Stack size=0x100 RAM=gpr14
#endif
#endif
```

13.6.1.3 C COMPILER SOURCE CODE

This is a simple code example showing the use of `#pragma romdata` for declaration of external memory and the use of C pointers for accessing this memory region.

If you are using MPLAB IDE to run this example, remember to enable external memory (`Configure>External Memory`) to see this extra memory in the Program Memory window.

```
#include <p18F8722.h>

// Microprocessor mode - a memory mode that supports external memory.
#pragma config MODE = MP

#pragma romdata SRAM_BASE // Assigns this romdata space at 0x20000
rom far char sram[];     // Declare an array at starting address

#pragma code
void main(void)
{
    rom far char* dataPtr; // Create a "far" pointer

    dataPtr = sram;        // Assign this pointer to the memory array
    *dataPtr++ = 0xCC;     // Write low byte of 16-bit word to SRAM
    *dataPtr = 0x55;       // Write high byte of 16-bit word to SRAM
}
```
13.6.2 Assembler Usage

This section discusses how to use the MPASM assembler (the assembler) when developing external memory application code.

In an assembler application's source file, using a simple #define or equ directive provides an easy way to define the SRAM starting address, which can be used to set up the table pointers prior to a table read or table write operation.

```assembly
#define SRAM_BASE_ADDRS 0x20000 ;Base addr for external ;memory device
#define SRAM_END_ADDRS  0x1FFFFF ;End addr (not required)
```

Accessing the external program memory through table reads and table writes requires the table pointer register be set up with the appropriate address as shown by the following example.

```assembly
movlw  upper (SRAM_BASE_ADDRS)
movwf  TBLPTRU
movlw  high (SRAM_BASE_ADDRS)
movwf  TBLPTRH
movlw  low (SRAM_BASE_ADDRS)
movwf  TBLPTRL
```

To build the assembler sample application, refer to Section 13.2 “How to Build the Sample Applications”.

13.6.2.1 ASSEMBLER MEMORY MAP

The figure below shows the memory mapping for the PIC18F8722 when used with the 1 Mbyte external SRAM device. Notice that the first 128K bytes of the external memory region is overlapped with the 128K bytes of internal program memory space and therefore cannot be accessed using the external memory bus. Without any additional external memory address decoding, the first 128K bytes of the SRAM are not accessible and therefore the first addressable location of SRAM is 20000h, as used in this example (SA4).

### TABLE 13-8: PROGRAM MEMORY MAP – PIC18F8722 AND 1 MB SRAM

<table>
<thead>
<tr>
<th>SA4</th>
<th>Program Memory Address</th>
<th>SRAM Address</th>
<th>Liner Script Section</th>
<th>Source Code Section</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x000000 0x000029</td>
<td>0x000000</td>
<td>vectors - Reset, Interrupts</td>
<td>vectors</td>
</tr>
<tr>
<td></td>
<td>0x00002A 0x01FFFF</td>
<td>0x01FFFF</td>
<td>page - On-chip Memory</td>
<td>prog - Main Program</td>
</tr>
<tr>
<td>→</td>
<td>0x020000 0x000000</td>
<td>0x020000</td>
<td>xsram - External Memory</td>
<td>SRAM_BASE_ADDRS</td>
</tr>
<tr>
<td></td>
<td>0x100000 0x1FFFFF</td>
<td>0x0000000</td>
<td></td>
<td>SRAM_END_ADDRS</td>
</tr>
</tbody>
</table>
13.6.2.2 ASSEMBLER LINKER SCRIPT

The modified PIC18F8722 assembler linker script file shown below demonstrates suggested modifications for external memory applications.

First the CODEPAGE statements are added.

CODEPAGE   NAME=vectors      START=0x0               END=0x29           PROTECTED
#IFDEF _DEBUGCODESTART
CODEPAGE   NAME=page       START=0x2A              END=_CODEEND
CODEPAGE   NAME=debug      START=_DEBUGCODESTART   END=_CEND        PROTECTED
#ELSE
CODEPAGE   NAME=page       START=0x2A              END=0x1FFFF           PROTECTED
#FI
CODEPAGE   NAME=xram       START=0x020000          END=0x1FFFFF       PROTECTED
CODEPAGE   NAME=idlos      START=0x200000          END=0x200007       PROTECTED
CODEPAGE   NAME=config     START=0x300000          END=0x30000D       PROTECTED
CODEPAGE   NAME=devid      START=0x3FFFFE          END=0x3FFFFF       PROTECTED
CODEPAGE   NAME=eedata     START=0xF00000          END=0xF003FF       PROTECTED
#IFDEF _CRUNTIME
SECTION    NAME=CONFIG     ROM=config
SECTION    NAME=VECTORS    ROM=vectors
SECTION    NAME=PROG       ROM=page
SECTION    NAME=SRAM       ROM=xram
#IFDEF _DEBUGDATASTART
STACK SIZE=0x100 RAM=gpr13
#ELSE
STACK SIZE=0x100 RAM=gpr14
#FI
#FI

Then the vectors, program, and external memory section is added.

#IFDEF _CRUNTIME
SECTION    NAME=CONFIG     ROM=config
SECTION    NAME=VECTORS    ROM=vectors
SECTION    NAME=PROG       ROM=page
SECTION    NAME=SRAM       ROM=xram
#IFDEF _DEBUGDATASTART
STACK SIZE=0x100 RAM=gpr13
#ELSE
STACK SIZE=0x100 RAM=gpr14
#FI
#FI
13.6.2.3 ASSEMBLER SOURCE CODE

This is a simple code example showing the definition of the external memory SRAM address at 20000h and how to write a 16-bit value to two consecutive memory locations using the table pointer register and table write instruction.

If you are using MPLAB IDE to run this example, remember to enable external memory (Configure>External Memory) to see this extra memory in the Program Memory window.

```
#include <p18F8722.inc>

; Microprocessor mode - a memory mode that supports external memory.
CONFIG MODE = MP

#define SRAM_BASE_ADDRS 0x20000    ; Base addr for external memory device
#define SRAM_END_ADDRS  0x1FFFFF   ; End addr (not required)

vectors code
  bra    main

prog   code
main:
  ; Example - how to write "0x55CC" to first word location in external SRAM memory

  movlw   upper (SRAM_BASE_ADDRS)
  movwf   TBLPTRU
  movlw   high (SRAM_BASE_ADDRS)
  movwf   TBLPTRH
  movlw   low (SRAM_BASE_ADDRS)
  movwf   TBLPTRL

  movlw   0xCC
  movwf   TABLAT
  tblwt*+          ; Writes "0xCC" to byte location 0x020000;
  ; Increments table pointer to next location

  movlw   0x55
  movwf   TABLAT
  tblwt*          ; Write "0x55" to byte location 0x020001;

END
```
14.1 INTRODUCTION

Error messages and warning messages are produced by the MPLINK linker. These messages always appear in the listing file directly above each line in which the error occurred.

Common problems and limitations of the linker tool are also listed here.

Topics covered in this chapter:
• Linker Parse Errors
• Linker Errors
• Linker Warnings
• COFF File Errors
• Other Errors, Warnings and Messages
• Common Problems

14.2 LINKER PARSE ERRORS

MPLINK linker parse errors are listed alphabetically below:

**Could not open 'cmdfile'**.
A linker command file could not be opened. Check that the file exists, is in the current search path, and is readable.

**Illegal <filename> for FILES in 'cmdfile:line'**.
An object or library filename must end with `.o` or `.lib` respectively.

**Illegal <filename> for INCLUDE in 'cmdfile:line'**.
A linker command filename must end with `.lkr`.

**Illegal <libpath> for LIBPATH in 'cmdfile:line'**.
The `libpath` must be a semicolon delimited list of directories. Enclose directory name which have embedded spaces in double quotes.

**Illegal <lkrpath> for LKRPATH in 'cmdfile:line'**.
The `lkrpath` must be a semicolon delimited list of directories. Enclose directory names which have embedded spaces in double quotes.

**Invalid attributes for memory in 'cmdfile:line'**.
A CODEPAGE, DATABANK, or SHAREBANK directive does not specify a NAME, START, or END attribute; or another attribute is specified which is not valid.

**Invalid attributes for SECTION in 'cmdfile:line'**.
A SECTION directive must have a NAME and either a RAM or ROM attribute.
Invalid attributes for STACK in 'cmdfile:line'.
A STACK directive does not specify a SIZE attribute, or another attribute is specified
which is not valid.

-k switch requires <pathlist>.
A semicolon delimited path must be specified. Enclose directory names containing
embedded spaces with double quotes. For example:
-k ..;c:\mylkr;"c:\program files\microchip\mpasm suite\lkr"

-l switch requires <pathlist>.
A semicolon delimited path must be specified. Enclose directory names containing
embedded spaces with double quotes. For example:
-l ..;c:\mylib;"c:\program files\microchip\mpasm suite"

-m switch requires <filename>.
A map filename must be specified. For example: -m main.map.

Multiple inclusion of library file 'filename'.
A library file has been included multiple times either on the command line or with a
FILES directive in a linker command file. Remove the multiple references.

Multiple inclusion of linker command file 'cmdfile'.
A linker command file can only be included once. Remove multiple INCLUDE directives
to the referenced linker command file.

Multiple inclusion of object file 'filename'.
An object file has been included multiple times either on the command line or with a
FILES directive in a linker command file. Remove the multiple references.

-n switch requires <length>.
The number of source lines per listing file page must be specified. A length of zero will
suppress pagination of the listing file.

-o switch requires <filename>.
A COFF output filename must be specified. For example: -o main.out

Unknown switch: 'cmdline token'.
An unrecognized command line switch was supplied. Refer to the Usage
documentation for the list of supported switches.

Unrecognized input in 'cmdfile:line'.
All statements in a linker command file must begin with a directive keyword or the
comment Delimiter //.
14.3 LINKER ERRORS

MPLINK linker errors are listed alphabetically below:

**Absolute code section 'secName' must start at a word-aligned address.**

Program code sections will only be allocated at word-aligned addresses. MPLINK will give this error message if an absolute code section address is specified that is not word-aligned.

**Configuration settings have been specified for address 0x300001 in more than one object module. Found in 'foo.o' previously found in 'bar.o'**

This error is issued when MPLAB C18's `#pragma config` directive has been used in two separate .c files (e.g., `foo.c` and `bar.c`) with settings specified from the same configuration byte. Set configuration bits for a given byte in a single .c file.

**Conflicting types for symbol 'symName'.**

Symbol `symName` is defined in different locations as different types.

**Could not find definition of symbol 'symName' in file 'filename'.**

A symbol `symName` is used without being defined in file `filename`.

**Could not find file 'filename'.**

An input object or library file was specified which does not exist, or cannot be found in the linker path.

**Could not open map file 'filename' for writing.**

Verify that if `filename` exists, it is not a read-only file.

**Could not resolve section reference 'symName' in file 'filename'.**

The symbol `symName` is an external reference. No input module defines this symbol. If the symbol is defined in a library module, ensure that the library module is included on the command line or in the linker command file using the FILES directive.

**Could not resolve symbol 'symName' in file 'filename'.**

The symbol `symName` is an external reference. No input module defines this symbol. If the symbol is defined in a library module, ensure that the library module is included on the command line or in the linker command file using the FILES directive.

**Duplicate definition of memory 'memName'.**

All CODEPAGE and DATABANK directives must have unique NAME attributes.

**Duplicate definitions of SECTION 'secName'.**

Each SECTION directive must have unique NAME attributes. Remove duplicate definitions.

**File ‘filename’, section ‘secName’, performs a call to symbol ‘symName’ which is not in the lower half of a page.**

For 12-bit devices, the program counter (PC), bit 8, is cleared in the CALL instruction or any modify PCL instruction. Therefore, all subroutine calls or computed jumps are limited to the first 256 locations of any program memory page (512 words long.)
Inconsistent length definitions of SHAREBANK 'memName'.
All SHAREBANK definitions which have the same NAME attribute must be of equal length.

Internal Coff output file is corrupt.
The linker cannot write to the COFF file.

Memory 'memName' overlaps memory 'memName'.
All CODEPAGE blocks must specify unique memory ranges which do not overlap. Similarly DATABANK and SHAREBANK blocks may not overlap.

Mixing extended and non-extended mode modules not allowed
The linker cannot link a mixture of extended mode modules and non-extended mode modules. Extended and non-extended memory modes apply to PIC18 devices.
When using MPASM to create object file modules, extended memory mode is enabled/disabled on the command line using the /y option. In MPLAB IDE, select Project>Build Options, MPASM Assembly tab, and check/uncheck the option "Extended Mode".
When using MPLAB C18 to create object file modules, extended memory mode is enabled/disabled on the command line using the --extended option. In MPLAB IDE, select Project>Build Options, MPLAB C18 tab, and check/uncheck the option "Extended Mode".
When using linker scripts, those with the suffix _e apply to extended mode use.

MPASM's __CONFIG directive (found in 'bar.o') cannot be used with either MPLAB C18's #pragma config directive or MPASM's CONFIG directive (found in 'foo.o')
This error message is issued when MPASM assembler's __CONFIG directive is specified in a .asm file (e.g., bar.asm) and MPLAB C18's #pragma config directive is specified in a .c file (e.g., foo.c). Set configuration bits using either MPASM assembler's __CONFIG directive or MPLAB C18's #pragma config directive.

Multiple map files declared: 'filename1', 'filename2'.
The -m <mapfile> switch was specified more than once.

Multiple output files declared: 'filename1', 'filename2'.
The -o <outfile> switch was specified more than once.

Multiple STACK definitions.
A STACK directive occurs more than once in the linker command file or included linker command files. Remove the multiple STACK directives.

No input object files specified.
No input object or library file was specified to the linker. Enter files to link.

Overlapping definitions of SHAREBANK 'memName'.
A SHAREBANK directive specifies a range of addresses that overlap a previous definition. Overlaps are not permitted.
{PCL | TOSH | TOSU | TOSL} cannot be used as the destination of a MOVFF or MOVSF instruction.

The MOVFF instruction has unpredictable results when its destination is the PCL, TOSH, TOSU, or TOSL registers. MPLINK will not allow the destination of a MOVFF instruction to be replaced with any of these addresses.

Processor types do not agree across all input files.

Each object module and library file specifies a processor type or a processor family. All input modules processor types or families must match.

Section {absolute|access|overlay|share} types for 'secName' do not match across input files.

A section with the name secName occurs in more than one input file. However, in some files it is marked as either an absolute, access, overlay or shared section, and in some it is not. Change the section's type in the source files and rebuild the object modules.

Section 'secName' can not fit the absolute section. Section 'secName' start=0xHHHH, length=0xHHHH.

A section which has not been assigned to a memory in the linker command file can not be allocated. Use the -m <mapfile> switch to generate an error map file. The error map will show the sections which were allocated prior to the error. More memory must be made available by adding a CODEPAGE, SHAREBANK, or DATABANK directive, or by removing the PROTECTED attribute, or the number of input sections must be reduced.

Section 'romName' can not have a 'RAM' memory attribute specified in the linker command file.

Use only the ROM attribute when defining the section in the linker command file.

Section 'secName' can not fit the section. Section 'secName' length='0xHHHH'.

A section which has not been assigned to a memory in the linker command file can not be allocated. Use the -m <mapfile> switch to generate an error map file. The error map will show the sections which were allocated prior to the error. More memory must be made available by adding a CODEPAGE, SHAREBANK, or DATABANK directive, or by removing the PROTECTED attribute, or the number of input sections must be reduced.

Section 'secName' contains code and can not have a 'RAM' memory attribute specified in the linker command file.

Use only the ROM attribute when defining the section in the linker command file.

Section 'secName' contains initialized data and can not have a 'ROM' memory attribute specified in the linker command file.

Use only the RAM attribute when defining the section in the linker command file.

Section 'secName' contains initialized rom data and can not have a 'RAM' memory attribute specified in the linker command file.

Use only the ROM attribute when defining the section in the linker command file.
Section 'secName' contains uninitialized data and can not have a 'ROM' memory attribute specified in the linker command file.

Use only the RAM attribute when defining the section in the linker command file.

Section 'secName' has a memory 'memName' which can not fit the absolute section. Section 'secName' start=0xHHHH, length=0xHHHH.

The memory which was assigned to the section in the linker command file either does not have space to fit the section, or the section will overlap another section. Use the -m <mapfile> switch to generate an error map file. The error map will show the sections which were allocated prior to the error.

Section 'secName' has a memory 'memName' which can not fit the section. Section 'secName' length='0xHHHH'.

The memory which was assigned to the section in the linker command file either does not have space to fit the section, or the section will overlap another section. Use the -m <mapfile> switch to generate an error map file. The error map will show the sections which were allocated prior to the error.

Section 'secName' has a memory 'memName' which is not defined in the linker command file.

Add a CODEPAGE, DATABANK, or SHAREBANK directive for the undefined memory to the linker command file.

Section 'secName' type is non-overlay and absolute but occurs in more than one input file.

An absolute section with the name secName may only occur in a single input file. Relocatable sections with the same name may occur in multiple input files. Either remove the multiple absolute sections in the source files or use relocatable sections instead.

Starting addresses for absolute overlay section 'secName' do not match across all input files.

A section with the name secName occurs in more than one input file. However, its absolute overlay starting address varies between files. Change the section's address in the source files and rebuild the object modules.

Symbol 'symName' has multiple definitions.

A symbol may only be defined in a single input module.

Symbol 'symName' is not word-aligned. It cannot be used as the target of a (branch | call or goto) instruction.

The target of a branch, call, or goto instruction was at an odd address, but the instruction encoding cannot reference addresses that are not word-aligned.

symbol 'symName' out of range of relative branch instruction.

A relative branch instruction had symName as its target, but a 2’s compliment encoding of the offset to symName wouldn't fit in the limited number of instruction bits used for the target of a branch instruction.
Errors, Warnings and Common Problems

The _CONFIG_DECL macro can only be specified in one module. Found in 'foo.o' previously found in 'bar.o'

This error is issued when MPLAB C18's _CONFIG_DECL macro is specified in two separate .c files (e.g., foo.c and bar.c). Set configuration bits by using the _CONFIG_DECL macro in only one .c file.

The _CONFIG_DECL macro (found in 'foo.o') cannot be used with MPASM's __CONFIG directive (found in 'bar.o')

This error is issued when MPLAB C18's _CONFIG_DECL macro is used in a .c file (e.g., foo.c) and MPASM assembler's __CONFIG directive is used in a .asm file (e.g., bar.asm). Set configuration bits using either the _CONFIG_DECL macro from MPLAB C18 or the __CONFIG directive in MPASM assembler.

The _CONFIG_DECL macro (found in 'foo.o') cannot be used with either MPLAB C18's #pragma config directive or MPASM's CONFIG directive (found in 'bar.o')

This error is issued when MPLAB C18's _CONFIG_DECL macro is used in a .c file (e.g., foo.c) with either MPLAB C18's #pragma config directive in a second .c file (e.g., bar.c) or MPASM assembler's __CONFIG directive in a .asm file (e.g., bar.asm). Set configuration bits by using only one of _CONFIG_DECL, #pragma config, or __CONFIG directive.

TRIS argument is out of range '0xHHHH' not between '0xHHHH' and '0xHHHH'.

Check the device data sheet to determine acceptable hex values for the TRIS register you are using.

Undefined CODEPAGE 'memName' for SECTION 'secName'.

A SECTION directive with a ROM attribute refers to a memory block which has not been defined. Add a CODEPAGE directive to the linker command file for the undefined memory block.

Undefined DATABANK/SHAREBANK 'memName' for SECTION 'secName'.

A SECTION directive with a RAM attribute refers to a memory block that has not been defined. Add a DATABANK or SHAREBANK directive to the linker command file for the undefined memory block.

Undefined DATABANK/SHAREBANK 'memName' for STACK.

No input object files specified. At least one object module must be specified either on the command line or in the linker command file using the FILES directive.

Unknown section type for 'secName'.

The section type for 'secName' needs to be defined.

Unknown section type for 'secName' in file 'filename'.

An input object or library module is not of the proper file type or it may be corrupted.

Unsupported processor type in file 'filename'.

A processor was specified that is not currently supported by the linker. See the Readme file for a list of supported devices.
Unsupported relocation type.

A relocation type was specified that is not currently supported by the linker.

### 14.4 LINKER WARNINGS

MPLINK linker warnings are listed alphabetically below:

**Fill pattern for memory 'memName' doesn't divide evenly into unused section locations. Last value was truncated.**

If a fill pattern is specified for a ROM section, but the free space in that section isn't evenly divisible by the fill pattern size, this warning will be issued to warn of incomplete patterns.

'/a' command line option ignored with '/x'

/x prevents the generation of a .hex file. Therefore, specifying the format of hex output file with /a is irrelevant.

'/n' command line option ignored with '/w'

/w prevents the generation of a .lst file. Therefore, specifying the number of lines per listing page with /n is irrelevant.

### 14.5 COFF FILE ERRORS

All the COFF errors listed below indicate an internal error in the file's contents. Please contact Microchip support if any of these errors are generated.

- Coff file 'filename' could not read file header.
- Coff file 'filename' could not read line numbers.
- Coff file 'filename' could not read optional file header.
- Coff file 'filename' could not read raw data.
- Coff file 'filename' could not read relocation info.
- Coff file 'filename' could not read section header.
- Coff file 'filename' could not read string table.
- Coff file 'filename' could not read string table length.
- Coff file 'filename' could not read symbol table.
- Coff file 'filename' could not write file header.
- Coff file 'filename' could not write lineinfo.
- Coff file 'filename' could not write optional file header.
- Coff file 'filename' could not write raw data.
- Coff file 'filename' could not write reloc.
- Coff file 'filename' could not write section header.
- Coff file 'filename' could not write string.
- Coff file 'filename' could not write string table length.
- Coff file 'filename' could not write symbol.
- Coff file 'filename', cScnHdr.size() != cScnNum.size().
- Coff file 'filename' does not appear to be a valid COFF file.
- Coff file 'filename' has relocation entries but an empty symbol table.
- Coff file 'filename' missing optional file header.
- Coff file 'filename' section["xx"] has an invalid s_offset.
- Coff file 'filename', section 'secName' line["xx"] has an invalid l_fcnndx.
- Coff file 'filename', section 'secName' line["xx"] has an invalid l_srcndx.
Errors, Warnings and Common Problems

- Coff file 'filename', section 'secName' reloc['xx'] has an invalid r_symndx.
- Coff file 'filename' symbol['xx'] has an invalid n_offset.
- Coff file 'filename' symbol['xx'] has an invalid n_scnum.
- Coff file 'filename', symbol['xx'] has an invalid index.
- Could not find section name 'secName' in string table.
- Could not find symbol name 'symName' in string table.
- Could not open Coff file 'filename' for reading.
- Could not open Coff file 'filename' for writing.
- Could not read archive magic string in library file 'filename'.
- Unable to find aux_file name in string table.
- Unable to find section name in string table.
- Unable to find symbol name in string table.

14.6 OTHER ERRORS, WARNINGS AND MESSAGES

If you are using the linker with any of the utilities, i.e., MPLIB librarian or you have not used the linker options /w or /x, then you may need to look in the utility troubleshooting sections for your error.

- MPLIB Librarian - Chapter 17. “Errors”
- MP2COD and/or MP2HEX - Chapter 19. “Errors and Warnings”

14.7 COMMON PROBLEMS

Although I set up listing file properties with MPASM assembler directives, none of these properties is appearing in my listing file.

Although MPASM assembler is often used with MPLINK object linker, MPASM assembler directives are not supported in MPLINK linker scripts. See Section 10.3 “Command Line Interface” for control of listing and hex file output.
Part 3 – MPLIB Object Librarian

Chapter 15. MPLIB Librarian Overview ................................................................. 235
Chapter 16. Librarian Interfaces ............................................................................ 239
Chapter 17. Errors ............................................................................................... 241
Chapter 15. MPLIB Librarian Overview

15.1 INTRODUCTION

An overview of the MPLIB object librarian and its capabilities is presented. Topics covered in this chapter:
- What is MPLIB Librarian
- How MPLIB Librarian Works
- How MPLIB Librarian Helps You
- Librarian Operation
- Librarian Input/Output Files

15.2 WHAT IS MPLIB LIBRARIAN

MPLIB object librarian (the librarian) combines object modules generated by the MPASM assembler or the MPLAB C18 C compiler into a single library file. This file may then be inputted into the MPLINK object linker.

15.3 HOW MPLIB LIBRARIAN WORKS

A librarian manages the creation and modification of library files. A library file is simply a collection of object modules that are stored in a single file. There are several reasons for creating library files:
- Libraries make linking easier. Since library files can contain many object files, the name of a library file can be used instead of the names of many separate object files when linking.
- Libraries help keep code small. Since a linker only uses the required object files contained in a library, not all object files which are contained in the library necessarily wind up in the linker's output module.
- Libraries make projects more maintainable. If a library is included in a project, the addition or removal of calls to that library will not require a change to the link process.
- Libraries help to convey the purpose of a group of object modules. Since libraries can group together several related object modules, the purpose of a library file is usually more understandable than the purpose of its individual object modules. For example, the purpose of a file named math.lib is more apparent than the purpose of power.o, ceiling.o, and floor.o.
15.4 HOW MPLIB LIBRARIAN HELPS YOU

The MPLIB librarian can help you in the following ways:

- The librarian makes linking easier because single libraries can be included instead of many smaller files.
- The librarian helps keep code maintainable by grouping related modules together.
- The librarian commands allow libraries to be created and modules to be added, listed, replaced, deleted, or extracted.

15.5 LIBRARIAN OPERATION

The librarian combines multiple input object modules, generated by the MPASM assembler or MPLAB C18 C compilers, into a single output library (.lib) file. Library files are used in conjunction with the MPLINK linker to produce executable code.

FIGURE 15-1: MPLIB™ LIBRARIAN OPERATION

![Diagram showing the process of combining object files into a library file.](image-url)
15.6 LIBRARIAN INPUT/OUTPUT FILES

The MPLIB librarian combines multiple object files into one library (.lib) file.

**Input Files**

| Object File (.o) | Relocatable code produced from source files. |

**Output Files**

| Library File (.lib) | A collection of object files grouped together for convenience. |

15.6.1 Object File (.o)

Object files are the relocatable code produced from source files. The MPLIB librarian combines several object files into a single library file.

15.6.2 Library File (.lib)

A library file may be created from object files by the MPLIB librarian or may be an existing standard library.
Chapter 16. Librarian Interfaces

16.1 INTRODUCTION

How to use MPLIB librarian is discussed here. For information on how librarian output can be used with the MPASM assembler and the MPLINK linker, see the documentation for these tools.

Topics covered in this chapter:
• MPLAB IDE Interface
• Command Line Options
• Command Line Examples and Tips

16.2 MPLAB IDE INTERFACE

The MPLIB librarian may be used with MPLAB IDE to create a library file from project object files instead of an executable (hex) file.

With your project open in MPLAB IDE, select Project>Build Options>Project. In the Build Options dialog, click on the MPASM/C17/C18 Suite tab. Click the radio button next to “Build library target (invoke MPLIB)”. Then click OK. Now when you build your project, you will be building a library file.
16.3 COMMAND LINE OPTIONS

MPLIB librarian is invoked with the following syntax:

```bash
mplib [/q] /{ctdrx} LIBRARY [MEMBER...]
```

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>/c</td>
<td>Create library</td>
<td>creates a new LIBRARY with the listed MEMBER(s)</td>
</tr>
<tr>
<td>/d</td>
<td>Delete member</td>
<td>deletes MEMBER(s) from the LIBRARY; if no MEMBER is specified the LIBRARY is not altered</td>
</tr>
<tr>
<td>/q</td>
<td>Quiet mode</td>
<td>no output is displayed</td>
</tr>
<tr>
<td>/r</td>
<td>Add/replace member</td>
<td>if MEMBER(s) exist in the LIBRARY, then they are replaced, otherwise MEMBER is appended to the end of the LIBRARY</td>
</tr>
<tr>
<td>/t</td>
<td>List members</td>
<td>prints a table showing the names of the members in the LIBRARY</td>
</tr>
<tr>
<td>/x</td>
<td>Extract member</td>
<td>if MEMBER(s) exist in the LIBRARY, then they are extracted. If no MEMBER is specified, all members will be extracted</td>
</tr>
</tbody>
</table>

16.4 COMMAND LINE EXAMPLES AND TIPS

**Example of Use**

Suppose you wanted to create a library named `dsp.lib` from three object modules named `fft.o`, `fir.o`, and `iir.o`. The following command line would produce the desired results:

```bash
mplib /c dsp.lib fft.o fir.o iir.o
```

To display the names of the object modules contained in a library file named `dsp.lib`, the following command line would be appropriate:

```bash
mplib /t dsp.lib
```

**Tips**

MPLIB librarian creates library files that may contain only a single external definition for any symbol. Therefore, if two object modules define the same external symbol, the librarian will generate an error if both object modules are added to the same library file.

To minimize the code and data space which results from linking with a library file, the library’s member object modules should be as small as possible. Creating object modules that contain only a single function can significantly reduce code space.
Chapter 17. Errors

17.1 INTRODUCTION

MPLIB librarian detects the following sources of error and reports them.

Topics covered in this chapter:

- Librarian Parse Errors
- Library File Errors
- COFF File Errors

17.2 LIBRARIAN PARSE ERRORS

MPLIB librarian parse errors are listed alphabetically below:

Invalid Object Filename

All object filenames must end with '.o'.

Invalid Switch

An unsupported switch was specified. For a list of supported switches, refer to command line options.

Library Filename is Required

All commands require a library filename. All library filenames must end with '.lib'.

17.3 LIBRARY FILE ERRORS

Library file processing errors are listed alphabetically below:

Could not build member 'memberName' in library file 'filename'.

The file is not a valid library file or it is corrupted.

Could not open library file 'filename' for reading.

Verify that filename exists and can be read.

Could not open library file 'filename' for writing.

Verify that if filename exists, it is not read-only.

Could not write archive magic string in library file 'filename'.

The file may be corrupted.

Could not write member header for 'memberName' in library file 'filename'.

The file may be corrupted.

File 'filename' is not a valid library file.

Library files must end with .lib.
Library file 'filename' has a missing member object file.
The file not a valid object file or it may be corrupted.

'memberName' is not a member of library 'filename'.

memberName can not be extracted or deleted from a library unless it is a member of the library.

Symbol 'symName' has multiple external definitions.
A symbol may only be defined once in a library file.

17.4 COFF FILE ERRORS

For a list of COFF File Errors, see MPLINK linker Section 14.5 “COFF File Errors”.
Chapter 18. Utilities Overview and Usage

18.1 INTRODUCTION

An overview of the 8-bit utilities and their capabilities are presented.

Topics covered in this chapter:
- What are Utilities
- Utilities Operation
- mp2hex.exe Utility
- mp2cod.exe Utility

18.2 WHAT ARE UTILITIES

Utilities are tools available for use with the assembler and/or linker. The MPLIB object librarian is a utility that was discussed in the previous sections.

<table>
<thead>
<tr>
<th>TABLE 18-1: AVAILABLE UTILITIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utility</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>mplib.exe</td>
</tr>
<tr>
<td>mp2hex.exe</td>
</tr>
<tr>
<td>mp2cod.exe</td>
</tr>
</tbody>
</table>

18.3 UTILITIES OPERATION

The utilities MP2HEX and MP2COD work with the MPLINK object linker to generate executable code (.hex) or a linker listing file (.lst) from the linker COF file. The Hex file is used by simulators, emulators, debuggers and programmers.

FIGURE 18-1: UTILITIES OPERATION

© 2009 Microchip Technology Inc.
18.4 MP2HEX.EXE UTILITY

Use this utility to take the MPLINK linker output COF file and create a Hex file. A Hex file contains no debug information but may be programmed directly into a device.

The MPLINK linker /x option will result in the linker not using this utility.

18.5 MP2COD.EXE UTILITY

Use this utility to take the MPLINK linker output COF file and create a COD file and a list file. A COD file is a legacy debug file that is no longer used (see Section 8.5.3 “MPASM Assembler Versions before v3.30”.) A list file generated by this utility is specific to the linker (see Section 9.7.6 “Listing File (.lst)”.)

The MPLINK linker /w option will result in the linker not using this utility.
Chapter 19. Errors and Warnings

19.1 INTRODUCTION

Error messages and warning messages are produced by the 8-bit utilities. These messages always appear in the listing file directly above each line in which the error occurred.

- Hex File Errors
- COFF To COD Conversion Errors
- COFF To COD Converter Warnings
- COD File Errors

19.2 HEX FILE ERRORS

Selected hex format does not support byte addresses above 64kb; use INHX32 format!

Your code addresses more than 64 Kbytes of program memory, but your selected hex format cannot support this. Switch to INHX32 format.

Could not open hex file 'filename' for writing.

The hex file was never created due to other errors, or an existing hex file is write-protected.

19.3 COFF TO COD CONVERSION ERRORS

Source file 'filename' name exceeds file format maximum of 63 characters.

The COD file name, including the path, has a 63-character limit.

Coff file 'filename' must contain at least one 'code' or 'romdata' section.

In order to convert a COFF file to a COD file, the COFF file must have either a code or a romdata section.

Could not open list file 'filename' for writing.

Verify that if filename exists and that it is not a read-only file.

19.4 COFF TO COD CONVERTER WARNINGS

Could not open source file 'filename'. This file will not be present in the list file.

The referenced source file could not be opened. This can happen if an input object/library module was built on a machine with a different directory structure. If source level debugging for the file is desired, rebuild the object or library on the current machine.
19.5  COD FILE ERRORS

All the COD file errors listed below indicate an internal error in the file's contents. Please contact Microchip support if any of these errors are generated.

- Cod file 'filename' does not have a proper debug message table.
- Cod file 'filename' does not have a proper Index.
- Cod file 'filename' does not have a proper line info table.
- Cod file 'filename' does not have a proper local vars table.
- Cod file 'filename' does not have a proper long symbol table.
- Cod file 'filename' does not have a proper memory map table.
- Cod file 'filename' does not have a proper name table.
- Cod file 'filename' does not have a proper symbol table.
- Cod file 'filename' does not have a properly formed first directory.
- Cod file 'filename' does not have a properly formed linked directory.
- Could not open Cod file ‘filename’ for reading.
- Could not open Cod file ‘filename’ for writing.
- Could not write 'blockname' block in Cod file ‘filename’.
- Could not write directory in Cod file ‘filename’.
Part 5 – Appendices

Appendix A. Instruction Sets ................................................................................................. 251
Appendix B. Useful Tables ...................................................................................................... 267
A.1 INTRODUCTION

PIC1X MCU instruction sets are used in developing applications with MPASM assembler, MPLINK object linker and MPLIB object librarian.

Instructions listed here are grouped either by instruction width or device number.

<table>
<thead>
<tr>
<th>Instruction Width</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Bit</td>
<td>PIC10F2XX, PIC12C5XX, PIC12CE5XX, PIC16X5X, PIC16C505</td>
</tr>
<tr>
<td>14 Bit</td>
<td>PIC12C67X, PIC12CE67X, PIC12F629/675, PIC16X</td>
</tr>
<tr>
<td>16 Bit</td>
<td>PIC18X</td>
</tr>
</tbody>
</table>

Topics covered are:
- Key to 12/14-Bit Instruction Width Instruction Sets
  - 12-Bit Instruction Width Instruction Set
  - 14-Bit Instruction Width Instruction Set
  - 14-Bit Instruction Width Extended Instruction Set
  - 12-Bit/14-Bit Instruction Width Pseudo-Instructions
- Key to PIC18 Device Instruction Set
  - PIC18 Device Instruction Set
  - PIC18 Device Extended Instruction Set

A.2 KEY TO 12/14-BIT INSTRUCTION WIDTH INSTRUCTION SETS

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination either the WREG register or the specified register file location. See d.</td>
</tr>
<tr>
<td>f</td>
<td>Register file address (5-bit, 7-bit or 8-bit).</td>
</tr>
<tr>
<td>n</td>
<td>FSR or INDF number (0 or 1).</td>
</tr>
<tr>
<td>p</td>
<td>Peripheral register file address (5-bit).</td>
</tr>
<tr>
<td>r</td>
<td>Port for TRIS.</td>
</tr>
<tr>
<td>x</td>
<td>Don't care (‘0′ or ‘1’). The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.</td>
</tr>
<tr>
<td>k</td>
<td>Literal field, constant data or label.</td>
</tr>
<tr>
<td>k</td>
<td>4-bit.</td>
</tr>
<tr>
<td>kk</td>
<td>8-bit.</td>
</tr>
<tr>
<td>kkk</td>
<td>12-bit.</td>
</tr>
<tr>
<td>mm</td>
<td>Pre-post increment-decrement mode selection.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>Bits</strong></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Bit address within an 8-bit file register (0 to 7).</td>
</tr>
</tbody>
</table>
| d     | Destination select bit.  
  d = 0: store result in WREG  
  d = 1: store result in file register f (default) |
| i     | Table pointer control.  
  i = 0: do not change.  
  i = 1: increment after instruction execution. |
| s     | Destination select bit.  
  s = 0: store result in file register f and WREG  
  s = 1: store result in file register f (default) |
| t     | Table byte select.  
  t = 0: perform operation on lower byte.  
  t = 1: perform operation on upper byte. |
| , ,   | Bit values, as opposed to Hex value. |

**Named Registers**

<table>
<thead>
<tr>
<th>Named Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR</td>
</tr>
<tr>
<td>OPTION</td>
</tr>
<tr>
<td>PCL, PCH</td>
</tr>
<tr>
<td>PCLATH, PCLATU</td>
</tr>
<tr>
<td>PRODH, PRODL</td>
</tr>
<tr>
<td>TBLATH, TBLATL</td>
</tr>
<tr>
<td>TBLPTR</td>
</tr>
<tr>
<td>WREG</td>
</tr>
</tbody>
</table>

**Named Bits**

<table>
<thead>
<tr>
<th>Named Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, DC, Z, OV, N</td>
</tr>
<tr>
<td>TØ</td>
</tr>
<tr>
<td>PD</td>
</tr>
<tr>
<td>GIE</td>
</tr>
</tbody>
</table>

**Named Device Features**

<table>
<thead>
<tr>
<th>Named Device Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
</tr>
<tr>
<td>TOS</td>
</tr>
<tr>
<td>WDT</td>
</tr>
</tbody>
</table>

**Misc. Descriptors**

<table>
<thead>
<tr>
<th>Misc. Descriptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
</tr>
<tr>
<td>→, ←</td>
</tr>
<tr>
<td>&lt; &gt;</td>
</tr>
</tbody>
</table>
A.3 12-BIT INSTRUCTION WIDTH INSTRUCTION SET

Microchip’s baseline 8-bit microcontroller family uses a 12-bit wide instruction set. All instructions execute in a single instruction cycle unless otherwise noted. Any unused opcode is executed as a NOP.

The instruction set is grouped into the following categories: byte-oriented file register operations, bit-oriented file register operations, and core literal and control operations. Instructions. Additionally, instructions that apply to both 12-bit and 14-bit devices are shown in Section A.6 “12-Bit/14-Bit Instruction Width Pseudo-Instructions”.

Instruction opcode is shown in hex by making certain assumptions, either listed in the key or as a footnote. For more information on the opcode bit values for each instruction, as well as the number of cycles per instruction, status bits affected and complete instruction details, see the relevant device data sheet.

### TABLE A-1: 12-BIT BYTE-ORIENTED FILE REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Ef*</td>
<td>ADDWF f,d</td>
<td>Add W and f</td>
<td>WREG + f → dest</td>
</tr>
<tr>
<td>16f*</td>
<td>ANDWF f,d</td>
<td>AND W and f</td>
<td>WREG .AND. f → dest</td>
</tr>
<tr>
<td>06f</td>
<td>CLRWF f</td>
<td>Clear f</td>
<td>0 → f</td>
</tr>
<tr>
<td>040</td>
<td>CLRW</td>
<td>Clear W</td>
<td>0 → WREG</td>
</tr>
<tr>
<td>26f*</td>
<td>COMWF f,d</td>
<td>Complement f</td>
<td>.NOT. f → dest</td>
</tr>
<tr>
<td>0Ef*</td>
<td>DECF f,d</td>
<td>Decrement f</td>
<td>f - 1 → dest</td>
</tr>
<tr>
<td>2Ef*</td>
<td>DECFSZ f,d</td>
<td>Decrement f, skip if zero</td>
<td>f - 1 → dest, skip if zero</td>
</tr>
<tr>
<td>2Af*</td>
<td>INCF f,d</td>
<td>Increment f</td>
<td>f + 1 → dest</td>
</tr>
<tr>
<td>3Ef*</td>
<td>INCFSZ f,d</td>
<td>Increment f, skip if zero</td>
<td>f + 1 → dest, skip if zero</td>
</tr>
<tr>
<td>12f*</td>
<td>IORWF f,d</td>
<td>Inclusive OR W and f</td>
<td>WREG .OR. f → dest</td>
</tr>
<tr>
<td>22f*</td>
<td>MOVWF f,d</td>
<td>Move f</td>
<td>f → dest</td>
</tr>
<tr>
<td>02f</td>
<td>MOVWF f</td>
<td>Move W to f</td>
<td>WREG → f</td>
</tr>
<tr>
<td>000</td>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>36f*</td>
<td>RLF f,d</td>
<td>Rotate left f</td>
<td></td>
</tr>
<tr>
<td>32f*</td>
<td>RRF f,d</td>
<td>Rotate right f</td>
<td></td>
</tr>
<tr>
<td>0Af*</td>
<td>SUBWF f,d</td>
<td>Subtract W from f</td>
<td>f - WREG → dest</td>
</tr>
<tr>
<td>3Af*</td>
<td>SWAPF f,d</td>
<td>Swap halves f</td>
<td>f(0:3) ← f(4:7) → dest</td>
</tr>
<tr>
<td>1Af*</td>
<td>XORWF f,d</td>
<td>Exclusive OR W and f</td>
<td>WREG .XOR. f → dest</td>
</tr>
</tbody>
</table>

* Assuming default bit value for d.

### TABLE A-2: 12-BIT BIT-ORIENTED FILE REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>4bf</td>
<td>BCF f,b</td>
<td>Bit clear f</td>
<td>0 → f(b)</td>
</tr>
<tr>
<td>5bf</td>
<td>BSF f,b</td>
<td>Bit set f</td>
<td>1 → f(b)</td>
</tr>
<tr>
<td>6bf</td>
<td>BTFSCL f,b</td>
<td>Bit test, skip if clear</td>
<td>skip if f(b) = 0</td>
</tr>
<tr>
<td>7bf</td>
<td>BTFSCH f,b</td>
<td>Bit test, skip if set</td>
<td>skip if f(b) = 1</td>
</tr>
</tbody>
</table>
A.4 14-BIT INSTRUCTION WIDTH INSTRUCTION SET

Microchip’s midrange 8-bit microcontroller family uses a 14-bit wide instruction set. This instruction set consists of 36 instructions, each a single 14-bit wide word. Most instructions operate on a file register, f, and the working register, WREG (accumulator). The result can be directed either to the file register or the WREG register or to both in the case of some instructions. A few instructions operate solely on a file register (BSF, for example).

The instruction set is grouped into the following categories: byte-oriented file register operations, bit-oriented file register operations, and core literal and control operations. Additionally, instructions that apply to both 12-bit and 14-bit devices are shown in Section A.6 “12-Bit/14-Bit Instruction Width Pseudo-Instructions”.

Instruction opcode is shown in hex by making certain assumptions, either listed in the key or as a footnote. For more information on the opcode bit values for each instruction, as well as the number of cycles per instruction, status bits affected and complete instruction details, see the relevant device data sheet.

### TABLE A-3: 12-BIT LITERAL AND CONTROL OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ekk</td>
<td>ANDLW kk</td>
<td>AND literal and W</td>
<td>kk .AND. WREG → WREG</td>
</tr>
<tr>
<td>9kk</td>
<td>CALL kk</td>
<td>Call subroutine</td>
<td>PC + 1 → TOS, kk → PC</td>
</tr>
<tr>
<td>004</td>
<td>CLRWDT</td>
<td>Clear watchdog timer</td>
<td>0 → WDT (and Prescaler if assigned)</td>
</tr>
<tr>
<td>Akk</td>
<td>GOTO kk</td>
<td>Goto address (k is nine bits)</td>
<td>kk → PC(9 bits)</td>
</tr>
<tr>
<td>Dkk</td>
<td>IORLW kk</td>
<td>Incl. OR literal and W</td>
<td>kk .OR. WREG → WREG</td>
</tr>
<tr>
<td>Ckk</td>
<td>MOVLW kk</td>
<td>Move Literal to W</td>
<td>kk → WREG</td>
</tr>
<tr>
<td>002</td>
<td>OPTION</td>
<td>Load OPTION Register</td>
<td>WREG → OPTION Register</td>
</tr>
<tr>
<td>8kk</td>
<td>RETLW kk</td>
<td>Return with literal in W</td>
<td>kk → WREG, TOS → PC</td>
</tr>
<tr>
<td>003</td>
<td>SLEEP</td>
<td>Go into Standby Mode</td>
<td>0 → WDT, stop oscillator</td>
</tr>
<tr>
<td>00r</td>
<td>TRIS r</td>
<td>Tristate port r</td>
<td>WREG → I/O control reg r</td>
</tr>
<tr>
<td>Fkk</td>
<td>XORLW kk</td>
<td>Exclusive OR literal and W</td>
<td>kk .XOR. WREG → WREG</td>
</tr>
</tbody>
</table>

### TABLE A-4: 14-BIT BYTE-ORIENTED FILE REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>07df</td>
<td>ADDWF f,d</td>
<td>Add W and f</td>
<td>W + f → d</td>
</tr>
<tr>
<td>05df</td>
<td>ANDWF f,d</td>
<td>AND W and f</td>
<td>W .AND. f → d</td>
</tr>
<tr>
<td>01'1'f</td>
<td>CLRF f</td>
<td>Clear f</td>
<td>0 → f</td>
</tr>
<tr>
<td>01xx</td>
<td>CLRW</td>
<td>Clear W</td>
<td>0 → W</td>
</tr>
<tr>
<td>09df</td>
<td>COMF f,d</td>
<td>Complement f</td>
<td>.NOT. f → d</td>
</tr>
<tr>
<td>03df</td>
<td>DECF f,d</td>
<td>Decrement f</td>
<td>f - 1 → d</td>
</tr>
<tr>
<td>0Bdf</td>
<td>DECFSZ f,d</td>
<td>Decrement f, skip if zero</td>
<td>f - 1 → d, skip if 0</td>
</tr>
<tr>
<td>0Adf</td>
<td>INCF f,d</td>
<td>Increment f</td>
<td>f + 1 → d</td>
</tr>
<tr>
<td>0Ddf</td>
<td>INCFSZ f,d</td>
<td>Increment f, skip if zero</td>
<td>f + 1 → d, skip if 0</td>
</tr>
<tr>
<td>04df</td>
<td>IORWF f,d</td>
<td>Inclusive OR W and f</td>
<td>W .OR. f → d</td>
</tr>
<tr>
<td>08df</td>
<td>MOVF f,d</td>
<td>Move f</td>
<td>f → d</td>
</tr>
<tr>
<td>00'1'f</td>
<td>MOVWF f</td>
<td>Move W to f</td>
<td>W → f</td>
</tr>
<tr>
<td>0000</td>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>0Ddf</td>
<td>RLF f,d</td>
<td>Rotate left f</td>
<td>register f</td>
</tr>
</tbody>
</table>
### TABLE A-4: 14-BIT BYTE-ORIENTED FILE REGISTER OPERATIONS (CONTINUED)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0Cdf</td>
<td>RRF</td>
<td>Rotate right f</td>
<td>$f \rightarrow f', d$</td>
</tr>
<tr>
<td>02df</td>
<td>SUBWF</td>
<td>Subtract W from f</td>
<td>$f - W \rightarrow d$</td>
</tr>
<tr>
<td>0Edf</td>
<td>SWAPPF</td>
<td>Swap halves f</td>
<td>$f(0:3) \leftrightarrow f(4:7) \rightarrow d$</td>
</tr>
<tr>
<td>06df</td>
<td>XORWF</td>
<td>Exclusive OR W and f</td>
<td>$W \text{ XOR } f \rightarrow d$</td>
</tr>
</tbody>
</table>

### TABLE A-5: 14-BIT BIT-ORIENTED FILE REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>4bf</td>
<td>BCF</td>
<td>Bit clear f</td>
<td>$0 \rightarrow f(b)$</td>
</tr>
<tr>
<td>5bf</td>
<td>BSF</td>
<td>Bit set f</td>
<td>$1 \rightarrow f(b)$</td>
</tr>
<tr>
<td>6bf</td>
<td>BTFSC</td>
<td>Bit test, skip if clear</td>
<td>skip if $f(b) = 0$</td>
</tr>
<tr>
<td>7bf</td>
<td>BTFSS</td>
<td>Bit test, skip if set</td>
<td>skip if $f(b) = 1$</td>
</tr>
</tbody>
</table>

### TABLE A-6: 14-BIT LITERAL AND CONTROL OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3Ekk</td>
<td>ADDLW</td>
<td>Add literal to W</td>
<td>$kk + WREG \rightarrow WREG$</td>
</tr>
<tr>
<td>39kk</td>
<td>ANDLW</td>
<td>AND literal and W</td>
<td>$kk \cdot \text{AND} \cdot WREG \rightarrow WREG$</td>
</tr>
<tr>
<td>2'0'kkk</td>
<td>CALL</td>
<td>Call subroutine</td>
<td>$PC + 1 \rightarrow TOS, kk \rightarrow PC$</td>
</tr>
<tr>
<td>0064</td>
<td>CLRWDT</td>
<td>Clear watchdog timer</td>
<td>$0 \rightarrow WDT$ (and Prescaler if assigned)</td>
</tr>
<tr>
<td>2'1'kkk</td>
<td>GOTO</td>
<td>Goto address (k is nine bits)</td>
<td>$kk \rightarrow PC(9 \text{ bits})$</td>
</tr>
<tr>
<td>38kk</td>
<td>IORLW</td>
<td>Incl. OR literal and W</td>
<td>$kk \cdot \text{OR} \cdot WREG \rightarrow WREG$</td>
</tr>
<tr>
<td>30kk</td>
<td>MOVWL</td>
<td>Move Literal to W</td>
<td>$kk \rightarrow WREG$</td>
</tr>
<tr>
<td>0062</td>
<td>OPTION</td>
<td>Load OPTION register</td>
<td>$WREG \rightarrow \text{OPTION Register}$</td>
</tr>
<tr>
<td>0009</td>
<td>RETFIE</td>
<td>Return from Interrupt</td>
<td>$TOS \rightarrow PC, 1 \rightarrow \text{GIE}$</td>
</tr>
<tr>
<td>34kk</td>
<td>RETLW</td>
<td>Return with literal in W</td>
<td>$kk \rightarrow WREG, TOS \rightarrow PC$</td>
</tr>
<tr>
<td>0008</td>
<td>RETURN</td>
<td>Return from subroutine</td>
<td>$TOS \rightarrow PC$</td>
</tr>
<tr>
<td>0063</td>
<td>SLEEP</td>
<td>Go into Standby Mode</td>
<td>$0 \rightarrow \text{WDT, stop oscillator}$</td>
</tr>
<tr>
<td>3Ckk</td>
<td>SUBLW</td>
<td>Subtract W from literal</td>
<td>$kk - WREG \rightarrow WREG$</td>
</tr>
<tr>
<td>006r</td>
<td>TRIS</td>
<td>Tristate port r</td>
<td>$WREG \rightarrow \text{I/O control reg } r$</td>
</tr>
<tr>
<td>3Akk</td>
<td>XORLW</td>
<td>Exclusive OR literal and W</td>
<td>$kk \cdot \text{XOR} \cdot WREG \rightarrow WREG$</td>
</tr>
</tbody>
</table>
A.5 14-BIT INSTRUCTION WIDTH EXTENDED INSTRUCTION SET

Some of Microchip's midrange 8-bit microcontroller family use a 14-bit wide extended instruction set. (Consult your device data sheet to see if you device uses an extended instruction set.) This instruction set consists of 41 instructions, each a single 14-bit wide word. Most instructions operate on a file register, f, and the working register, WREG (accumulator). The result can be directed either to the file register or the WREG register or to both in the case of some instructions. A few instructions operate solely on a file register (BSF, for example).

The instruction set is grouped into the following categories: byte-oriented file register operations, byte-oriented skip operations, bit-oriented file register operations, bit-oriented skip operations, core literal operations, core control operations, core inherent operations and C-compiler optimized operations. Additionally, instructions that apply to both 12-bit and 14-bit devices are shown in Section A.6 “12-Bit/14-Bit Instruction Width Pseudo-Instructions”.

Instruction opcode is show in hex by making certain assumptions, either listed in the key or as a footnote. For more information on the opcode bit values for each instruction, as well as the number of cycles per instruction, status bits affected and complete instruction details, see the relevant device data sheet.

**TABLE A-7: 14-BIT BYTE-ORIENTED FILE REGISTER OPERATIONS**

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>07df</td>
<td>ADDWF f,d</td>
<td>Add W and f</td>
<td>W + f → d</td>
</tr>
<tr>
<td>3Ddf</td>
<td>ADDWF C f,d</td>
<td>Add with Carry W and f(1)</td>
<td>W + f + C → d</td>
</tr>
<tr>
<td>05df</td>
<td>ANDWF f,d</td>
<td>AND W with f</td>
<td>W .AND. f → d</td>
</tr>
<tr>
<td>37df</td>
<td>ASRF f,d</td>
<td>Arithmatic Right Shift(1)</td>
<td>register f</td>
</tr>
<tr>
<td>35df</td>
<td>LSLF f,d</td>
<td>Logical Left Shift(1)</td>
<td>register f</td>
</tr>
<tr>
<td>36df</td>
<td>LSRF f,d</td>
<td>Logical Right Shift(1)</td>
<td>register f</td>
</tr>
<tr>
<td>01'1'f</td>
<td>CLRF f</td>
<td>Clear f</td>
<td>0 → f</td>
</tr>
<tr>
<td>01xx</td>
<td>CLRW</td>
<td>Clear W</td>
<td>0 → W</td>
</tr>
<tr>
<td>09df</td>
<td>COMF f,d</td>
<td>Complement f</td>
<td>.NOT. f → d</td>
</tr>
<tr>
<td>03df</td>
<td>DECf f,d</td>
<td>Decrement f</td>
<td>f - 1 → d</td>
</tr>
<tr>
<td>0Adf</td>
<td>INCf f,d</td>
<td>Increment f</td>
<td>f + 1 → d</td>
</tr>
<tr>
<td>04df</td>
<td>IORWF f,d</td>
<td>Inclusive OR W with f</td>
<td>W .OR. f → d</td>
</tr>
<tr>
<td>08df</td>
<td>MOVf f,d</td>
<td>Move f</td>
<td>f → d</td>
</tr>
<tr>
<td>00'1'f</td>
<td>MOVWF f</td>
<td>Move W to f</td>
<td>W → f</td>
</tr>
<tr>
<td>0Ddf</td>
<td>RLF f,d</td>
<td>Rotate left f</td>
<td>register f</td>
</tr>
<tr>
<td>0Cdf</td>
<td>RRf f,d</td>
<td>Rotate right f</td>
<td>register f</td>
</tr>
</tbody>
</table>

(1) Except CLRF, CLRW, and CLRW.
### TABLE A-7: 14-BIT BYTE-ORIENTED FILE REGISTER OPERATIONS (CONTINUED)

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>02df</td>
<td>SUBWF f,d</td>
<td>Subtract W from f</td>
<td>f - W → d</td>
</tr>
<tr>
<td>3Bdf</td>
<td>SUBWFB f,d</td>
<td>Subtract with Borrow W from ( f(1) )</td>
<td>f - W - B → d</td>
</tr>
<tr>
<td>0Edf</td>
<td>SWAPF f,d</td>
<td>Swap halves f</td>
<td>( f(0:3) ↔ f(4:7) ) → d</td>
</tr>
<tr>
<td>06df</td>
<td>XORWF f,d</td>
<td>Exclusive OR W and f</td>
<td>W .XOR. f → d</td>
</tr>
</tbody>
</table>

**Note 1:** Operation in 14-bit extended instruction set but not 14-bit instruction set.

### TABLE A-8: 14-BIT BYTE-ORIENTED SKIP OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0Bdf</td>
<td>DECFSZ f,d</td>
<td>Decrement f, skip if zero</td>
<td>f - 1 → d, skip if 0</td>
</tr>
<tr>
<td>0Fdf</td>
<td>INCFSZ f,d</td>
<td>Increment f, skip if zero</td>
<td>f + 1 → d, skip if 0</td>
</tr>
</tbody>
</table>

### TABLE A-9: 14-BIT BYTE-ORIENTED FILE REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>4bf</td>
<td>BCF f,b</td>
<td>Bit clear f</td>
<td>0 → f(b)</td>
</tr>
<tr>
<td>5bf</td>
<td>BSF f,b</td>
<td>Bit set f</td>
<td>1 → f(b)</td>
</tr>
</tbody>
</table>

### TABLE A-10: 14-BIT BIT-ORIENTED SKIP OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>6bf</td>
<td>BTFSC f,b</td>
<td>Bit test, skip if clear</td>
<td>skip if f(b) = 0</td>
</tr>
<tr>
<td>7bf</td>
<td>BTFSS f,b</td>
<td>Bit test, skip if set</td>
<td>skip if f(b) = 1</td>
</tr>
</tbody>
</table>

### TABLE A-11: 14-BIT LITERAL OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3Ekk</td>
<td>ADDLW kk</td>
<td>Add literal to W</td>
<td>kk + WREG → WREG</td>
</tr>
<tr>
<td>39kk</td>
<td>ANDLW kk</td>
<td>AND literal and W</td>
<td>kk .AND. WREG → WREG</td>
</tr>
<tr>
<td>38kk</td>
<td>IORLW kk</td>
<td>Incl. OR literal and W</td>
<td>kk .OR. WREG → WREG</td>
</tr>
<tr>
<td>002k</td>
<td>MOVLB k</td>
<td>Move literal to BSR(^{(1)})</td>
<td>k → BSR</td>
</tr>
<tr>
<td>31'1'kk</td>
<td>MOVLP kk</td>
<td>Move literal to PCLATH(^{(1)})</td>
<td>kk → PCLATH</td>
</tr>
<tr>
<td>30kk</td>
<td>MOVLW kk</td>
<td>Move Literal to W</td>
<td>kk → WREG</td>
</tr>
<tr>
<td>3Ckk</td>
<td>SUBLW kk</td>
<td>Subtract W from literal</td>
<td>kk - WREG → WREG</td>
</tr>
<tr>
<td>3Akk</td>
<td>XORLW kk</td>
<td>Exclusive OR literal and W</td>
<td>kk .XOR. WREG → WREG</td>
</tr>
</tbody>
</table>

**Note 1:** Operation in 14-bit extended instruction set but not 14-bit instruction set.

### TABLE A-12: 14-BIT CONTROL OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>32kk</td>
<td>BRA kk</td>
<td>Relative branch(^{(1)})</td>
<td>PC + kk → PC</td>
</tr>
<tr>
<td>000B</td>
<td>BRW</td>
<td>Relative branch with W(^{(1)})</td>
<td>PC + W → PC</td>
</tr>
<tr>
<td>2'0'kkk</td>
<td>CALL kk</td>
<td>Call subroutine</td>
<td>PC + 1 → TOS, kkk → PC</td>
</tr>
<tr>
<td>000A</td>
<td>CALLW</td>
<td>Call subroutine with W(^{(1)})</td>
<td>PC + 1 → TOS, W → PC</td>
</tr>
<tr>
<td>2'1'kkk</td>
<td>GOTO kkk</td>
<td>Goto address (k is nine bits)</td>
<td>kk → PC(9 bits)</td>
</tr>
<tr>
<td>0009</td>
<td>RETFIE</td>
<td>Return from Interrupt</td>
<td>TOS → PC, 1 → GIE</td>
</tr>
<tr>
<td>34kk</td>
<td>RETLW k</td>
<td>Return with literal in W</td>
<td>kk → WREG, TOS → PC</td>
</tr>
<tr>
<td>0008</td>
<td>RETURN</td>
<td>Return from subroutine</td>
<td>TOS → PC</td>
</tr>
</tbody>
</table>

**Note 1:** Operation in 14-bit extended instruction set but not 14-bit instruction set.
### TABLE A-13: 14-BIT INHERENT OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0064</td>
<td>CLRWDT</td>
<td>Clear watchdog timer</td>
<td>$0 \rightarrow$ WDT (and Prescaler if assigned)</td>
</tr>
<tr>
<td>0000</td>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>0062</td>
<td>OPTION</td>
<td>Load OPTION register</td>
<td>WREG $\rightarrow$ OPTION Register</td>
</tr>
<tr>
<td>0001</td>
<td>RESET</td>
<td>Software device reset(^{(1)})</td>
<td>TOS $\rightarrow$ PC</td>
</tr>
<tr>
<td>0063</td>
<td>SLEEP</td>
<td>Go into Standby Mode</td>
<td>$0 \rightarrow$ WDT, stop oscillator</td>
</tr>
<tr>
<td>006r</td>
<td>TRIS r</td>
<td>Tristate port r</td>
<td>WREG $\rightarrow$ I/O control reg r</td>
</tr>
</tbody>
</table>

**Note 1:** Operation in 14-bit extended instruction set but not 14-bit instruction set.

### TABLE A-14: 14-BIT C-COMpiler OPTIMIZED OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31'0'nk</td>
<td>ADDFSR n, k</td>
<td>Add Literal to FSRn(^{(1)})</td>
<td>FSR(n) + k $\rightarrow$ FSR(n)</td>
</tr>
<tr>
<td>001'0'mn 001'0'nmm 3F'0'nk</td>
<td>MOVIW mm n, n mm k[n]</td>
<td>Move INDFn to W, with pre/post inc/dec(^{(1)})</td>
<td>INDFn $\rightarrow$ W</td>
</tr>
<tr>
<td>001'1'mn 001'1'nmm 3F'1'nk</td>
<td>MOVWI mm n, n mm k[n]</td>
<td>Move W to INDFn, with pre/post inc/dec(^{(1)})</td>
<td>W $\rightarrow$ INDFn</td>
</tr>
</tbody>
</table>

**Note 1:** Operation in 14-bit extended instruction set but not 14-bit instruction set.
### A.6 12-BIT/14-BIT INSTRUCTION WIDTH PSEUDO-INSTRUCTIONS

The following pseudo-instructions are applicable to both the 12-bit and 14-bit instruction word devices. These pseudo-instructions are alternative mnemonics for standard PIC1X instructions or are macros that generate one or more PIC1X instructions. Use of these pseudo-instructions is not recommended for new designs. These are documented mainly for historical purposes.

#### TABLE A-15: 12-BIT/14-BIT SPECIAL INSTRUCTION MNEMONICS

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Equivalent Operation(s)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDCF</td>
<td>Add Carry to File</td>
<td>BTFSC 3,0 f,d INCF f,d</td>
<td>Z</td>
</tr>
<tr>
<td>ADDDCF</td>
<td>Add Digit Carry to File</td>
<td>BTFSC 3,1 f,d INCF f,d</td>
<td>Z</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>BC</td>
<td>Branch on Carry</td>
<td>BTFSC 3,0 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>BDC</td>
<td>Branch on Digit Carry</td>
<td>BTFSC 3,1 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>BNC</td>
<td>Branch on No Carry</td>
<td>BTFSS 3,0 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>BNDC</td>
<td>Branch on No Digit Carry</td>
<td>BTFSS 3,1 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>BNZ</td>
<td>Branch on No Zero</td>
<td>BTFSS 3,2 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>BZ</td>
<td>Branch on Zero</td>
<td>BTFSS 3,2 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>CLRC</td>
<td>Clear Carry</td>
<td>BCF 3,0</td>
<td>-</td>
</tr>
<tr>
<td>CLRDCC</td>
<td>Clear Digit Carry</td>
<td>BCF 3,1</td>
<td>-</td>
</tr>
<tr>
<td>CLRZ</td>
<td>Clear Zero</td>
<td>BCF 3,2</td>
<td>-</td>
</tr>
<tr>
<td>LCALL</td>
<td>Long Call</td>
<td>BCF/BSF 0x0A,3 0x0A,4 CALL k</td>
<td>-</td>
</tr>
<tr>
<td>LGOTO</td>
<td>Long GOTO</td>
<td>BCF/BSF 0x0A,3 0x0A,4 GOTO k</td>
<td>-</td>
</tr>
<tr>
<td>MOVFW</td>
<td>Move File to W</td>
<td>MOVF f,0</td>
<td>Z</td>
</tr>
<tr>
<td>NEGF</td>
<td>Negate File</td>
<td>COMF f,1 INCF f,d</td>
<td>Z</td>
</tr>
<tr>
<td>SETC</td>
<td>Set Carry</td>
<td>BSF 3,0</td>
<td>-</td>
</tr>
<tr>
<td>SETDC</td>
<td>Set Digit Carry</td>
<td>BSF 3,1</td>
<td>-</td>
</tr>
<tr>
<td>SETZ</td>
<td>Set Zero</td>
<td>BSF 3,2</td>
<td>-</td>
</tr>
<tr>
<td>SKPC</td>
<td>Skip on Carry</td>
<td>BTFSS 3,0</td>
<td>-</td>
</tr>
<tr>
<td>SKPDC</td>
<td>Skip on Digit Carry</td>
<td>BTFSS 3,1</td>
<td>-</td>
</tr>
<tr>
<td>SKPNC</td>
<td>Skip on No Carry</td>
<td>BTFSS 3,0</td>
<td>-</td>
</tr>
<tr>
<td>SKPNDC</td>
<td>Skip on No Digit Carry</td>
<td>BTFSS 3,1</td>
<td>-</td>
</tr>
<tr>
<td>SKPNZ</td>
<td>Skip on Non Zero</td>
<td>BTFSS 3,2</td>
<td>-</td>
</tr>
<tr>
<td>SKPZ</td>
<td>Skip on Zero</td>
<td>BTFSS 3,2</td>
<td>-</td>
</tr>
<tr>
<td>SUBCF</td>
<td>Subtract Carry from File</td>
<td>BTFSS 3,0 DECF f,d</td>
<td>Z</td>
</tr>
</tbody>
</table>
A.7 KEY TO PIC18 DEVICE INSTRUCTION SET

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>Destination either the WREG register or the specified register file location. See d.</td>
</tr>
<tr>
<td>f</td>
<td>Register file address.</td>
</tr>
<tr>
<td>f'</td>
<td>8-bit (0x00 to 0xFF).</td>
</tr>
<tr>
<td>f''</td>
<td>12-bit (0x000 to 0xFFF). This is the source address.</td>
</tr>
<tr>
<td>f''</td>
<td>12-bit (0x000 to 0xFFF). This is the destination address.</td>
</tr>
<tr>
<td>x</td>
<td>0, 1 or 2 for FSR number.</td>
</tr>
<tr>
<td>x'</td>
<td>Don't care (&quot;0' or '1').</td>
</tr>
<tr>
<td>z</td>
<td>Indirect addressing offset.</td>
</tr>
<tr>
<td>z'</td>
<td>7-bit offset value for indirect addressing of register files (source).</td>
</tr>
<tr>
<td>z''</td>
<td>7-bit offset value for indirect addressing of register files (destination).</td>
</tr>
<tr>
<td>k</td>
<td>Literal field, constant data or label.</td>
</tr>
<tr>
<td>k’</td>
<td>4-bit.</td>
</tr>
<tr>
<td>k’k</td>
<td>8-bit.</td>
</tr>
<tr>
<td>k’k’k</td>
<td>12-bit.</td>
</tr>
<tr>
<td>n</td>
<td>The relative address (2's complement number) for relative branch instructions, or the direct address for Call/Branch and Return instructions.</td>
</tr>
<tr>
<td>*</td>
<td>The mode of the TBLPTR register for the table read and table write instructions.</td>
</tr>
<tr>
<td>*+</td>
<td>No Change to register</td>
</tr>
<tr>
<td>*-</td>
<td>Post-Increment register</td>
</tr>
<tr>
<td>+*</td>
<td>Post-Decrement register</td>
</tr>
<tr>
<td>+*</td>
<td>Pre-Increment register</td>
</tr>
</tbody>
</table>

Offsets, Increments/Decrements

Bits

<table>
<thead>
<tr>
<th>a</th>
<th>RAM access bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 0</td>
<td>RAM location in Access RAM (BSR register is ignored)</td>
</tr>
<tr>
<td>a = 1</td>
<td>RAM bank is specified by BSR register (default)</td>
</tr>
<tr>
<td>b</td>
<td>Bit address within an 8-bit file register (0 to 7).</td>
</tr>
</tbody>
</table>

Named Registers

<table>
<thead>
<tr>
<th>BSR</th>
<th>Bank Select Register. Used to select the current RAM bank.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSR</td>
<td>File Select Register.</td>
</tr>
<tr>
<td>PCL</td>
<td>Program Counter Low Byte.</td>
</tr>
</tbody>
</table>
A.8 PIC18 DEVICE INSTRUCTION SET

Microchip’s new high-performance 8-bit microcontroller family uses a 16-bit wide instruction set. This instruction set consists of 76 instructions, each a single 16-bit wide word (2 bytes). Most instructions operate on a file register, f, and the working register, WREG (accumulator). The result can be directed either to the file register or the WREG register or to both in the case of some instructions. A few instructions operate solely on a file register (BSF, for example).

The instruction set is grouped into the following categories: byte-oriented file register operations, bit-oriented file register operations, control operations, literal operations and memory operations. Additionally, extended mode instructions are shown in Section A.9 “PIC18 Device Extended Instruction Set”.

Instruction opcode is shown in hex by making certain assumptions, either listed in the key or as a footnote. For more information on the opcode bit values for each instruction, as well as the number of cycles per instruction, status bits affected and complete instruction details, see the relevant device data sheet.

### TABLE A-16: PIC18 BYTE-ORIENTED REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>27f*</td>
<td>ADDWF</td>
<td>ADD WREG to f</td>
<td>WREG+f → dest</td>
</tr>
<tr>
<td>23f*</td>
<td>ADDWFPC</td>
<td>ADD WREG and Carry bit to f</td>
<td>WREG+f+C → dest</td>
</tr>
<tr>
<td>17f*</td>
<td>ANDWF</td>
<td>AND WREG with f</td>
<td>WREG .AND. f → dest</td>
</tr>
<tr>
<td>6Bf*</td>
<td>CLRF</td>
<td>Clear f</td>
<td>0 → f</td>
</tr>
<tr>
<td>Hex</td>
<td>Mnemonic</td>
<td>Mnemonic</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>1Ff*</td>
<td>COMF</td>
<td>f,d,a</td>
<td>Complement f</td>
</tr>
<tr>
<td>63f*</td>
<td>CPFSEQ</td>
<td>f,a</td>
<td>Compare f with WREG, skip if f=WREG</td>
</tr>
<tr>
<td>65f*</td>
<td>CPFSGT</td>
<td>f,a</td>
<td>Compare f with WREG, skip if f &gt; WREG</td>
</tr>
<tr>
<td>61f*</td>
<td>CPFSLT</td>
<td>f,a</td>
<td>Compare f with WREG, skip if f &lt; WREG</td>
</tr>
<tr>
<td>07f*</td>
<td>DBCF</td>
<td>f,d,a</td>
<td>Decrement f</td>
</tr>
<tr>
<td>2Ff*</td>
<td>DECFSZ</td>
<td>f,d,a</td>
<td>Decrement f, skip if 0</td>
</tr>
<tr>
<td>4Ff*</td>
<td>DCFSNZ</td>
<td>f,d,a</td>
<td>Decrement f, skip if not 0</td>
</tr>
<tr>
<td>2Bf*</td>
<td>INCF</td>
<td>f,d,a</td>
<td>Increment f</td>
</tr>
<tr>
<td>3Ff*</td>
<td>INCFSZ</td>
<td>f,d,a</td>
<td>Increment f, skip if 0</td>
</tr>
<tr>
<td>4Bf*</td>
<td>INFSNZ</td>
<td>f,d,a</td>
<td>Increment f, skip if not 0</td>
</tr>
<tr>
<td>13f*</td>
<td>IORWF</td>
<td>f,d,a</td>
<td>Inclusive OR WREG with f</td>
</tr>
<tr>
<td>53f*</td>
<td>MOVF</td>
<td>f,d,a</td>
<td>Move f</td>
</tr>
<tr>
<td>CFf'</td>
<td>MOVFF</td>
<td>f',f&quot;</td>
<td>Move f' to f&quot; (second word)</td>
</tr>
<tr>
<td>6Ff*</td>
<td>MOVWF</td>
<td>f,a</td>
<td>Move WREG to f</td>
</tr>
<tr>
<td>03f*</td>
<td>MULWF</td>
<td>f,a</td>
<td>Multiply WREG with f</td>
</tr>
<tr>
<td>6Df*</td>
<td>NBGF</td>
<td>f,a</td>
<td>Negate f</td>
</tr>
<tr>
<td>37f*</td>
<td>RLCF</td>
<td>f,d,a</td>
<td>Rotate left f through Carry</td>
</tr>
<tr>
<td>47f*</td>
<td>RLNCF</td>
<td>f,d,a</td>
<td>Rotate left f (no carry)</td>
</tr>
<tr>
<td>33f*</td>
<td>RRCF</td>
<td>f,d,a</td>
<td>Rotate right f through Carry</td>
</tr>
<tr>
<td>43f*</td>
<td>RRNCF</td>
<td>f,d,a</td>
<td>Rotate right f (no carry)</td>
</tr>
<tr>
<td>69f*</td>
<td>SETF</td>
<td>f,a</td>
<td>Set f</td>
</tr>
<tr>
<td>57f*</td>
<td>SUBFWB</td>
<td>f,d,a</td>
<td>Subtract f from WREG with Borrow</td>
</tr>
<tr>
<td>5Ff*</td>
<td>SUBWF</td>
<td>f,d,a</td>
<td>Subtract WREG from f</td>
</tr>
<tr>
<td>5Bf*</td>
<td>SUBWFB</td>
<td>f,d,a</td>
<td>Subtract WREG from f with Borrow</td>
</tr>
<tr>
<td>3Bf*</td>
<td>SWAPF</td>
<td>f,d,a</td>
<td>Swap nibbles of f</td>
</tr>
<tr>
<td>67f*</td>
<td>TSTFSZ</td>
<td>f,a</td>
<td>Test f, skip if 0</td>
</tr>
<tr>
<td>1Bf*</td>
<td>XORWF</td>
<td>f,d,a</td>
<td>Exclusive OR WREG with f</td>
</tr>
</tbody>
</table>

* Assuming default bit values for d and a.
### TABLE A-17: PIC18 BIT-ORIENTED REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>91f*</td>
<td>BCF f, b, a</td>
<td>Bit Clear f</td>
<td>0 → (f_b)</td>
</tr>
<tr>
<td>81f*</td>
<td>BSF f, b, a</td>
<td>Bit Set f</td>
<td>1 → (f_b)</td>
</tr>
<tr>
<td>B1f*</td>
<td>BTFSC f, b, a</td>
<td>Bit test f, skip if clear</td>
<td>if (f_b = 0), PC+4 → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>A1f*</td>
<td>BTFSS f, b, a</td>
<td>Bit test f, skip if set</td>
<td>if (f_b = 1), PC+4 → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>71f*</td>
<td>BTG f, b, a</td>
<td>Bit Toggle f</td>
<td>(f_b \rightarrow f_b)</td>
</tr>
</tbody>
</table>

* Assuming \(b = 0\) and default bit value for a.

### TABLE A-18: PIC18 CONTROL OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2n</td>
<td>BC n</td>
<td>Branch if Carry</td>
<td>if (C=1), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>E6n</td>
<td>BN n</td>
<td>Branch if Negative</td>
<td>if (N=1), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>E3n</td>
<td>BNC n</td>
<td>Branch if Not Carry</td>
<td>if (C=0), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>E7n</td>
<td>BNN n</td>
<td>Branch if Not Negative</td>
<td>if (N=0), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>E5n</td>
<td>BNOV n</td>
<td>Branch if Not Overflow</td>
<td>if (OV=0), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>E1n</td>
<td>BNZ n</td>
<td>Branch if Zero</td>
<td>if (Z=0), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>E4n</td>
<td>BOV n</td>
<td>Branch if Overflow</td>
<td>if (OV=1), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>D’0’n</td>
<td>BRA n</td>
<td>Branch Unconditionally</td>
<td>PC+2+2*n → PC</td>
</tr>
<tr>
<td>E0n</td>
<td>BZ n</td>
<td>Branch if Zero</td>
<td>if (Z=1), PC+2+2*n → PC, else PC+2 → PC</td>
</tr>
<tr>
<td>ECkk*</td>
<td>CALL n, s</td>
<td>Call Subroutine</td>
<td>1st word PC+4 → TOS, n → PC&lt;20:1&gt;, 2nd word if s=1, WREG → WREGs, STATUS → STATUSs, BSR → BSRs</td>
</tr>
<tr>
<td>0004</td>
<td>CLRWDT</td>
<td>Clear Watchdog Timer</td>
<td>0 → WDT, 0 → WDT postscaler, 1 → TO,1 → PD</td>
</tr>
<tr>
<td>0007</td>
<td>DAW</td>
<td>Decimal Adjust WREG</td>
<td>if WREG&lt;3:0&gt; &gt;9 or DC=1, WREG&lt;3:0&gt;+6 → WREG&lt;3:0&gt;, else WREG&lt;3:0&gt; → WREG&lt;3:0&gt;; if WREG&lt;7:4&gt; &gt;9 or C=1, WREG&lt;7:4&gt;+6 → WREG&lt;7:4&gt;, else WREG&lt;7:4&gt; → WREG&lt;7:4&gt;;</td>
</tr>
<tr>
<td>EFkk</td>
<td>GOTO n</td>
<td>Go to address</td>
<td>1st word n → PC&lt;20:1&gt;, 2nd word</td>
</tr>
<tr>
<td>0000</td>
<td>NOP</td>
<td>No Operation</td>
<td>No Operation</td>
</tr>
<tr>
<td>Fxxx</td>
<td>NOP</td>
<td>No Operation</td>
<td>No Operation (2-word instructions)</td>
</tr>
<tr>
<td>0006</td>
<td>POP</td>
<td>Pop top of return stack (TOS)</td>
<td>TOS-1 → TOS</td>
</tr>
<tr>
<td>0005</td>
<td>PUSH</td>
<td>Push top of return stack (TOS)</td>
<td>PC +2 → TOS</td>
</tr>
<tr>
<td>D’1’n</td>
<td>RCALL n</td>
<td>Relative Call</td>
<td>PC+2 → TOS, PC+2+2*n→PC</td>
</tr>
<tr>
<td>00FF</td>
<td>RESET</td>
<td>Software device reset</td>
<td>Same as MCLR reset</td>
</tr>
<tr>
<td>0010*</td>
<td>RETFIE s</td>
<td>Return from interrupt (and enable interrupts)</td>
<td>TOS → PC, 1 → GIE/GIEH or PEIE/GIEL, if s=1, WREGs → WREG, STATUSs → STATUS, BSRs → BSR, PCLATU/PCLATH unchngd.</td>
</tr>
<tr>
<td>0012*</td>
<td>RETURN s</td>
<td>Return from subroutine</td>
<td>TOS → PC, if s=1, WREGs → WREG, STATUSs → STATUS, BSRs → BSR, PCLATU/PCLATH are unchanged</td>
</tr>
<tr>
<td>0003</td>
<td>SLEEP</td>
<td>Enter SLEEP Mode</td>
<td>0 → WDT, 0 → WDT postscaler, 1 → TO, 0 → PD</td>
</tr>
</tbody>
</table>

* Assuming default bit value for s.
### TABLE A-19: PIC18 LITERAL OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0Fkk</td>
<td>ADDLW kk</td>
<td>Add literal to WREG</td>
<td>WREG + kk → WREG</td>
</tr>
<tr>
<td>0Bkk</td>
<td>ANDLW kk</td>
<td>AND literal with WREG</td>
<td>WREG .AND. kk → WREG</td>
</tr>
<tr>
<td>09kk</td>
<td>IORLW kk</td>
<td>Inclusive OR literal with WREG</td>
<td>WREG .OR. kk → WREG</td>
</tr>
<tr>
<td>EErk</td>
<td>LFSR r,kk</td>
<td>Move literal (12 bit) 2nd word to FSRr 1st word</td>
<td>kk → FSRr</td>
</tr>
<tr>
<td>010k</td>
<td>MOVLB k</td>
<td>Move literal to BSR&lt;3:0&gt;</td>
<td>kk → BSR</td>
</tr>
<tr>
<td>0Ekk</td>
<td>MOVLW kk</td>
<td>Move literal to WREG</td>
<td>kk → WREG</td>
</tr>
<tr>
<td>0Dkk</td>
<td>MULLW kk</td>
<td>Multiply literal with WREG</td>
<td>WREG * kk → PRODH:PRODL</td>
</tr>
<tr>
<td>0Ckk</td>
<td>RETLW kk</td>
<td>Return with literal in WREG</td>
<td>kk → WREG</td>
</tr>
<tr>
<td>08kk</td>
<td>SUBLW kk</td>
<td>Subtract WREG from literal</td>
<td>kk−WREG → WREG</td>
</tr>
<tr>
<td>0Akk</td>
<td>XORLW kk</td>
<td>Exclusive OR literal with WREG</td>
<td>WREG .XOR. kk → WREG</td>
</tr>
</tbody>
</table>

### TABLE A-20: PIC18 MEMORY OPERATIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0008</td>
<td>TBLRD*</td>
<td>Table Read</td>
<td>Prog Mem (TBLPTR) → TABLAT</td>
</tr>
<tr>
<td>0009</td>
<td>TBLRD*+</td>
<td>Table Read with post-increment</td>
<td>Prog Mem (TBLPTR) → TABLAT TBLPTR +1 → TBLPTR</td>
</tr>
<tr>
<td>000A</td>
<td>TBLRD*-</td>
<td>Table Read with post-decrement</td>
<td>Prog Mem (TBLPTR) → TABLAT TBLPTR -1 → TBLPTR</td>
</tr>
<tr>
<td>000B</td>
<td>TBLRD++</td>
<td>Table Read with pre-increment</td>
<td>TBLPTR +1 → TBLPTR Prog Mem (TBLPTR) → TABLAT</td>
</tr>
<tr>
<td>000C</td>
<td>TBLWT*</td>
<td>Table Write</td>
<td>TABLAT → Prog Mem(TBLPTR)</td>
</tr>
<tr>
<td>000D</td>
<td>TBLWT*+</td>
<td>Table Write with post-increment</td>
<td>TABLAT → Prog Mem(TBLPTR) TBLPTR +1 → TBLPTR</td>
</tr>
<tr>
<td>000E</td>
<td>TBLWT*-</td>
<td>Table Write with post-decrement</td>
<td>TABLAT → Prog Mem(TBLPTR) TBLPTR -1 → TBLPTR</td>
</tr>
<tr>
<td>000F</td>
<td>TBLWT++</td>
<td>Table Write with pre-increment</td>
<td>TBLPTR +1 → TBLPTR TABLAT → Prog Mem(TBLPTR)</td>
</tr>
</tbody>
</table>
A.9  PIC18 DEVICE EXTENDED INSTRUCTION SET

Some PIC18 devices have an extended mode of operation for use with the MPLAB C18 compiler. This mode will change the operation of some instructions listed in Section A.8 “PIC18 Device Instruction Set” and add the instructions listed in this section.

In general, you should not need to use the extended instruction set. However, if needed, the extended mode is set using a special device configuration bit. For more on extended mode, see the *MPLAB C18 C Compiler User’s Guide* (DS51288) and your device data sheet.

Instruction opcode is shown in hex by making certain assumptions, either listed in the key or as a footnote. For more information on the opcode bit values for each instruction, as well as the number of cycles per instruction, status bits affected and complete instruction details, see the relevant device data sheet.

### TABLE A-21: PIC18 EXTENDED INSTRUCTIONS

<table>
<thead>
<tr>
<th>Hex</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>E8fk</td>
<td>ADDFSR f,k</td>
<td>Add literal to FSR</td>
<td>(FSR(f)+k → FSR(f))</td>
</tr>
<tr>
<td>E8Ck</td>
<td>ADDULNK k</td>
<td>Add literal to FSR2 and return</td>
<td>FSR2+k → FSR2, (TOS) → PC</td>
</tr>
<tr>
<td>0014</td>
<td>CALLW</td>
<td>Call subroutine using WREG</td>
<td>(PC + 2) → TOS, (W) → PCL, (PCLATH) → PCH, (PCLATU) → PCU</td>
</tr>
<tr>
<td>EB’0’z Pfff</td>
<td>MOVSF z’,f”</td>
<td>Move z’ (source) to 1st word, f” (destination)2nd word</td>
<td>((FSR2)+z’) → f”</td>
</tr>
<tr>
<td>EB’1’z Pxzz</td>
<td>MOVSS z’,z”</td>
<td>Move z’ (source) to 1st word, z” (destination)2nd word</td>
<td>((FSR2)+z’) → ((FSR2)+z”)</td>
</tr>
<tr>
<td>EAkk</td>
<td>PUSHL k</td>
<td>Store literal at FSR2, decrement FSR2</td>
<td>k → (FSR2), FSR2-1 → FSR2</td>
</tr>
<tr>
<td>E9fk</td>
<td>SUBFSR f,k</td>
<td>Subtract literal from FSR</td>
<td>FSR(f-k) → FSR(f)</td>
</tr>
<tr>
<td>E9Ck</td>
<td>SUBULNK k</td>
<td>Subtract literal from FSR2 and return</td>
<td>FSR2-k → FSR2, (TOS) → PC</td>
</tr>
</tbody>
</table>
Appendix B. Useful Tables

B.1 INTRODUCTION

Some useful tables are included for reference here. The tables are:

- ASCII Character Set
- Hexadecimal to Decimal Conversion

B.2 ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>Least Significant Nibble</th>
<th>HEX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Most Significant Nibble</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>NUL</td>
<td>DLE</td>
<td>Space</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Bell</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>CR</td>
<td>GS</td>
<td>–</td>
<td>=</td>
<td>M</td>
<td>}</td>
<td>m</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
<td></td>
</tr>
</tbody>
</table>
### B.3 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.

<table>
<thead>
<tr>
<th>High Byte</th>
<th>Low Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HEX 1000</strong></td>
<td><strong>Dec</strong></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4096</td>
</tr>
<tr>
<td>2</td>
<td>8192</td>
</tr>
<tr>
<td>3</td>
<td>12288</td>
</tr>
<tr>
<td>4</td>
<td>16384</td>
</tr>
<tr>
<td>5</td>
<td>20480</td>
</tr>
<tr>
<td>6</td>
<td>24576</td>
</tr>
<tr>
<td>7</td>
<td>28672</td>
</tr>
<tr>
<td>8</td>
<td>32768</td>
</tr>
<tr>
<td>9</td>
<td>36864</td>
</tr>
<tr>
<td>A</td>
<td>40960</td>
</tr>
<tr>
<td>B</td>
<td>45056</td>
</tr>
<tr>
<td>C</td>
<td>49152</td>
</tr>
<tr>
<td>D</td>
<td>53248</td>
</tr>
<tr>
<td>E</td>
<td>57344</td>
</tr>
<tr>
<td>F</td>
<td>61440</td>
</tr>
</tbody>
</table>

For example, HEX A38F converts to 41871 as follows:

<table>
<thead>
<tr>
<th>HEX 1000’s Digit</th>
<th>HEX 100’s Digit</th>
<th>HEX 10’s Digit</th>
<th>HEX 1’s Digit</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>40960</td>
<td>768</td>
<td>128</td>
<td>15</td>
<td>41871  Decimal</td>
</tr>
</tbody>
</table>
Glossary

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

Access Memory (PIC18 Only)
Special registers on PIC18 devices that allow access regardless of the setting of the bank select register (BSR).

Address
Value that identifies a location in memory.

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

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

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

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

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

Archiver
A tool that creates and manipulates libraries.

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

Assembler
A language tool that translates assembly language source code into machine code.

Assembly Language
A programming language that describes binary machine code in a symbolic form.

Asynchronous Stimulus
Data generated to simulate external inputs to a simulator device.

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.

**Calibration Memory**
A special function register or registers used to hold values for calibration of a PIC1X microcontroller on-board RC oscillator or other device peripherals.

**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.

**Configuration Bits**
Special-purpose bits programmed to set PIC1X microcontroller modes of operation. A configuration bit may or may not be preprogrammed.

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

**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 DSC devices.

**Directives**
Statements in source code that provide control of the language tool’s operation.

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

**DSC**

See Digital Signal Controller.

**EEPROM**

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

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

**Emulation**

The process of executing software loaded into emulation memory as if it were firmware residing on a microcontroller device.

**Emulation Memory**

Program memory contained within the emulator.

**Emulator**

Hardware that performs emulation.

**Emulator System**

The MPLAB ICE 2000 and 4000 emulator systems include the pod, processor module, device adapter, cables, and MPLAB IDE software.

**Environment - IDE**

The particular layout of the desktop for application development.

**Environment - MPLAB PM3**

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

**EPROM**

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

**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.

**Export**

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

**Extended Microcontroller Mode**

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC17 or PIC18 device.

**External Label**

A label that has external linkage.

**External Linkage**

A function or variable has external linkage if it can be referenced from outside the module in which it is defined.
External Symbol
A symbol for an identifier which has external linkage. This may be a reference or a definition.

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

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

External RAM
Off-chip Read/Write memory.

File Registers
On-chip data memory, including general purpose registers (GPRs) and special function registers (SFRs).

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

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

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

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

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

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.

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

ICD
In-Circuit Debugger. MPLAB ICD 2 is Microchip’s in-circuit debugger.

ICE
In-Circuit Emulator. MPLAB ICE 2000 and 4000 are Microchip’s in-circuit emulators.

IDE
Integrated Development Environment. MPLAB IDE is Microchip’s integrated development environment.

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

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

**Interrupt Service Routine**
User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

**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.

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

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

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.

Make Project
A command that rebuilds an application, re-compiling only those source files that have changed since the last complete compilation.

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.

Microcontroller Mode
One of the possible program memory configurations of PIC17 and PIC18 microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode
One of the possible program memory configurations of PIC17 and PIC18 microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

**Mnemonics**

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

**MPASM Assembler**

Microchip Technology’s relocatable macro assembler for PIC1X 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 PIC17 devices and MPLAB C18 is the C compiler for PIC18 devices.

**MPLAB C30**

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

**MPLAB ICD 2**

Microchip’s in-circuit debugger that works with MPLAB IDE. The ICD supports Flash devices with built-in debug circuitry. 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 ICE 2000/4000**

Microchip’s in-circuit emulators that works with MPLAB IDE. MPLAB ICE 2000 supports PIC1X MCUs. MPLAB ICE 4000 supports PIC18F MCUs and dsPIC30F DSCs. The main component of each ICE is the pod. A complete system consists of a pod, processor module, cables, and MPLAB IDE Software.

**MPLAB IDE**

Microchip’s Integrated Development Environment.

**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 PM3**

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

**MPLAB SIM**

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

**MPLIB Object Librarian**

MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler 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.

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

**Nesting Depth**
The maximum level to which macros can include other macros.

**Node**
MPLAB IDE project component.

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

**Non-Volatile Storage**
A storage device whose contents are preserved when its power is off.

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

**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.

**Object File Directives**
Directives that are used only when creating an object file.

**Off-Chip Memory**
Off-chip memory refers to the memory selection option for the PIC17 or PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the Emulator.

**Opcodes**
Operational Codes. See Mnemonics.

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

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

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

PC Host
Any IBM or compatible personal computer running a supported Windows operating system.

PIC MCUs
PIC microcontrollers (MCUs) refer to all Microchip microcontroller families.
PIC1X MCUs refer to 8-bit PIC10/12/16/18 devices, excluding 16-bit PIC24 devices.

PICSTART Plus
A developmental device programmer from Microchip. Programs 8-, 14-, 28-, and 40-pin PIC1X microcontrollers. Must be used with MPLAB IDE Software.

Pod, Emulator
The external emulator box that contains emulation memory, trace memory, event and cycle timers, and trace/breakpoint logic.

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

PRO MATE II
A device programmer from Microchip. Programs most PIC1X microcontrollers as well as most memory and Keeloq devices. Can be used with MPLAB IDE or stand-alone.

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

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

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

Project
A set of source files and instructions to build the object and executable code for an application.

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

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

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

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.
Real-Time
When released from the halt state in the emulator or MPLAB ICD mode, the processor runs in real-time mode and behaves exactly as the normal chip would behave. In real-time mode, the real-time trace buffer of MPLAB ICE is enabled and constantly captures all selected cycles, and all break logic is enabled. In the emulator or MPLAB ICD, the processor executes in real-time until a valid breakpoint causes a halt, or until the user halts the emulator. In the simulator real-time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

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

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

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

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

SFR
See Special Function Registers.

Shell
The MPASM assembler shell is a prompted input interface to the macro assembler.

Simulator
A software program that models the operation of devices.

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

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

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

Source Code
The form in which a computer program is written by the programmer. Source code is written in 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 PIC microcontroller where the return address is stored when a function call is made.

**Stack, Software**
Memory used by an application for storing return addresses, function parameters, and local variables. This memory is typically managed by the compiler when developing code in a high-level language.

**Static RAM or SRAM**
Static Random Access Memory. Program memory you can Read/Write on the target board that does not need refreshing frequently.

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

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

**Step Over**
Step Over allows you to step over subroutines. This command executes the code in the subroutine and then stops execution at the return address to the subroutine.

When stepping over a CALL instruction, the next breakpoint will be set at the instruction after the CALL. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. Select Halt to regain control of program execution.

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

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

**Stopwatch**
A counter for measuring execution cycles.

**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.

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

Target
Refers to user hardware.

Target Application
Software residing on the target board.

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

Target Processor
The microcontroller device on the target application board.

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

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

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

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

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

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

```c
int myVar;
```

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

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

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

Watch Variable
A variable that you may monitor during a debugging session in a watch window.
Watch Window
Watch windows contain a list of watch variables that are updated at each breakpoint.

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

WDT
See Watchdog Timer.

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

#### Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>__badram</td>
<td>46</td>
</tr>
<tr>
<td>__badrom</td>
<td>47</td>
</tr>
<tr>
<td>__config</td>
<td>55, 146</td>
</tr>
<tr>
<td>__fuses</td>
<td>55</td>
</tr>
<tr>
<td>__idlocs</td>
<td>85, 146</td>
</tr>
<tr>
<td>__maxram</td>
<td>95</td>
</tr>
<tr>
<td>__maxrom</td>
<td>96</td>
</tr>
<tr>
<td>__CRUNTIME</td>
<td>188</td>
</tr>
<tr>
<td>__DEBUG</td>
<td>188</td>
</tr>
<tr>
<td>__DEBUGCODELEN</td>
<td>188</td>
</tr>
<tr>
<td>__DEBUGCODESTART</td>
<td>188</td>
</tr>
<tr>
<td>__DEBUGDATALLEN</td>
<td>188</td>
</tr>
<tr>
<td>__DEBUGDASTART</td>
<td>188</td>
</tr>
<tr>
<td>__EXTENDEDMODE</td>
<td>188</td>
</tr>
<tr>
<td>__mplink.exe</td>
<td>196</td>
</tr>
<tr>
<td>.asm</td>
<td>12</td>
</tr>
<tr>
<td>.c</td>
<td>12</td>
</tr>
<tr>
<td>.cof</td>
<td>12</td>
</tr>
<tr>
<td>.hex</td>
<td>12</td>
</tr>
<tr>
<td>.lib</td>
<td>12</td>
</tr>
<tr>
<td>.lkr</td>
<td>12, 174</td>
</tr>
<tr>
<td>.o</td>
<td>12</td>
</tr>
<tr>
<td>/o</td>
<td>148</td>
</tr>
<tr>
<td>#define</td>
<td>65</td>
</tr>
<tr>
<td>#include</td>
<td>90, 141</td>
</tr>
<tr>
<td>#undefined</td>
<td>116</td>
</tr>
<tr>
<td>$</td>
<td>40</td>
</tr>
</tbody>
</table>

#### A

**Absolute Code, Generating** | 24
---|---
**Access Section**
- Overlayed | 46
**ACCESSBANK** | 183, 186
**Accessing Labels From Other Modules** | 146
**Allocation**
- Absolute | 192
- Relocatable | 192
- Stack | 192
**AND, logical** | 41
**Arithmetic Operators** | 40
**ASCII Character Set** | 267

#### B

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>badram</td>
<td>46</td>
</tr>
<tr>
<td>badrom</td>
<td>47</td>
</tr>
<tr>
<td>Bank Selecting</td>
<td>50</td>
</tr>
<tr>
<td>Bank Selecting, Indirect</td>
<td>48</td>
</tr>
<tr>
<td>Banking</td>
<td>128, 147</td>
</tr>
<tr>
<td>bankisel</td>
<td>48</td>
</tr>
<tr>
<td>banksel</td>
<td>50, 147</td>
</tr>
</tbody>
</table>

#### Bit Assignments

<table>
<thead>
<tr>
<th>Bit Assignments</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Assignments</td>
<td>127</td>
</tr>
</tbody>
</table>

#### Blank Listing Lines

<table>
<thead>
<tr>
<th>Blank Listing Lines</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank Listing Lines</td>
<td>109</td>
</tr>
</tbody>
</table>

#### Block of Constants

<table>
<thead>
<tr>
<th>Block of Constants</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block of Constants</td>
<td>52, 69</td>
</tr>
</tbody>
</table>

#### Boot Loader

<table>
<thead>
<tr>
<th>Boot Loader</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boot Loader</td>
<td>206</td>
</tr>
</tbody>
</table>

#### Build Options

<table>
<thead>
<tr>
<th>Build Options</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Options</td>
<td>13</td>
</tr>
</tbody>
</table>

#### Build Project

**Build Project**
- Command Line | 198
- MPLAB IDE | 196

#### C

**Caveats, Linker Script** | 183
---|---
**cblock** | 52
**COD file** | 167
**code** | 54, 142, 146
**Code Section** | 54, 127
**Code Section, Packed** | 55
**code_pack** | 55
**Code, Absolute** | 24, 142, 145, 147, 148
**Code, Relocatable** | 25, 142, 145, 147
**Calling File** | 149
**Defining Module** | 146
**Library Routine** | 149
**Referencing Module** | 146
**CODEPAGE** | 185, 186
**COF File** | 12
**COFF Object Module File** | 174
**Command Line Interface**
- Assembler | 35
- Librarian | 240
- Linker | 179

#### Command Line Options, Librarian

<table>
<thead>
<tr>
<th>Command Line Options, Librarian</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>/c</td>
<td>240</td>
</tr>
<tr>
<td>/d</td>
<td>240</td>
</tr>
<tr>
<td>/q</td>
<td>240</td>
</tr>
<tr>
<td>/r</td>
<td>240</td>
</tr>
<tr>
<td>/t</td>
<td>240</td>
</tr>
<tr>
<td>/x</td>
<td>240</td>
</tr>
</tbody>
</table>

#### Comments

<table>
<thead>
<tr>
<th>Comments</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>28</td>
</tr>
</tbody>
</table>

#### Common Problems

**Common Problems**
- Linker | 231
- Compiler | 12, 14, 15
**Conditional Assembly Directives** | 44
- else | 68
- endif | 70
- endw | 71
- fi | 70
- if | 86
- ifdef | 88
- ifndef | 89
- while | 118

#### Conditional Linker Statements

<table>
<thead>
<tr>
<th>Conditional Linker Statements</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional Linker Statements</td>
<td>187</td>
</tr>
<tr>
<td>config</td>
<td>57</td>
</tr>
</tbody>
</table>
Configuration Bits ............................................ 55, 57, 146
constant ................................................................ 58
Constant Compare .................................... 154
Constants
  Block Of .................................................. 52, 69
  Declare .................................................. 58
  Define .................................................. 71
  Control Directives .................................. 44
    #define ............................................. 65
    #include ........................................... 90
    #undef ............................................. 116
    constant ........................................... 58
    end ............................................... 69
    equ ............................................... 71
    org ................................................ 99
    processor ......................................... 104
    radix ............................................. 105
    set ............................................... 108
    variable ........................................ 117
Create Numeric and Text Data ............... 60
Customer Notification Service ............. 6
Customer Support .................................... 7

D
da ....................................................... 59
Data
  Byte .................................................. 62
  EEPROM Byte ...................................... 64
  Word .................................................. 68
data ................................................... 37, 60
Data Directives ........................................ 44
  #badram ......................................... 46
  #badrom ......................................... 47
  #config .......................................... 55
  #fuses .......................................... 55
  #idlocs .......................................... 85
  #maxram ......................................... 95
  #maxrom ......................................... 96
  cblock ........................................... 52
  config ............................................. 57
da ................................................... 59
data .................................................. 60
db ................................................... 62
de ........................................................ 64
dt ..................................................... 67
dtm .................................................. 67
dw ................................................... 68
endc ................................................. 69
fill .................................................. 80
res .................................................. 106
Data Section
  Access Uninitialized ................................ 111
  Initialized ......................................... 82
  Initialized Access .................................. 84
  Overlayed Uninitialized ............................ 113
  Shared Uninitialized ................................ 115
  Uninitialized ..................................... 110
Data, Initialized ........................................ 194
DATABANK ............................................. 183, 186
db ................................................... 62
de ................................................... 64

Debug
  Command Line ...................................... 189
  MPLAB IDE ......................................... 181
  Decrement ......................................... 41
  define ............................................ 65
  Delete a Substitution Label .................. 116
  Directives ........................................ 27
  Directives, Assembler ......................... 43
  Directives, Linker ................................ 182
  Documentation
    Conventions ...................................... 4
    Layout ........................................... 1
dt ................................................... 67
dtm .................................................. 67
dw ................................................... 37, 68

E
EEPROM .................................................. 203
EEPROM Data Byte ................................... 64
Eight-by-Eight Multiply ....................... 153
else ................................................. 68
end .................................................. 69
endc ............................................... 69
endif ............................................... 70
endm ................................................. 70
endw ................................................. 71
Environment Variables ....................... 198, 199
equ .................................................. 71
error .................................................. 72
Error File ........................................... 30, 155
errorlevel .......................................... 73
Errors
  Assembler ......................................... 155
  COFF .............................................. 230
  COFF to COD Converter ....................... 247
  Librarian Parse .................................. 241
  Linker ............................................ 225
  Linker Parse ..................................... 223
Escape Sequences .................................... 38
Examples, Application
  #define ........................................... 66, 130, 134
  #include .......................................... 122
  #undef ........................................... 66, 130
  banksel .......................................... 49, 50
  banksel .......................................... 51, 52, 123, 132
  cblock ........................................... 53
  code ............................................... 54, 123
  constant ........................................ 118, 130
da ................................................... 59
data .................................................. 61
de ................................................... 63
de ................................................... 65
else .................................................. 87
end ................................................. 122
endc ............................................... 53
endif ............................................... 87
endm ................................................. 94, 132
endw .................................................. 119
equ .................................................. 108, 123
error .................................................. 72
errorlevel ......................................... 74
Examples, Simple
__badram......................................................47
__badrom........................................................48
__config..........................................................56
__idlocs............................................................85
__maxram.......................................................47
__maxrom........................................................48
#define............................................................66
#include................................................................91
#undef..............................................................117
bankisel..........................................................48
banksel...........................................................50
cblock.............................................................53
code.................................................................54
code_pack........................................................55
cconfig............................................................58
data.................................................................60
db.................................................................63
de.................................................................64
dt.................................................................67
dw.................................................................68
dwelse............................................................69
dwend.............................................................69
dendm...............................................................70
equ.................................................................72
error..............................................................72
errorlevel........................................................74
exitm..............................................................76
extern............................................................78
fill.............................................................80, 81
global.............................................................82
idata..........................................................83, 85
if.................................................................87
ifdef.........................................................88, 89
ifndef..........................................................89
ifdef...............................................................88
if...............................................................86
else..............................................................68
end.................................................................70
ifdef.............................................................88
IFDEF/ELSE/FI in Linker Scripts........................187
ifndef............................................................89
INCLUDE......................................................183
include..........................................................90
Include Additional Source File..........................90
messg...........................................................97
org..................................................................99
pagesize.......................................................102, 104
processor.......................................................104
radix.............................................................105
res..............................................................106, 130
set...............................................................108, 130
udata.............................................................110
udata_acs.......................................................112
udata_ova.......................................................113
udata_sh.......................................................115
variable.........................................................118, 130
while.............................................................119

F
fi.................................................................70
File
   Error..................................................................155
   Listing...................................................................45
FILES..............................................................182
fill...............................................................80
Final Frontier.................................................109

G
Generic Linker Script Example.............................189
Generic Linker Scripts........................................200
global............................................................82

H
Header Files...................................................90, 126, 141
Hex Files.........................................................12, 30, 174
Hexadecimal to Decimal Conversion......................268
high..................................................................40, 143

I
ID Locations..................................................85, 146
idata.............................................................82, 145
idata_acs........................................................84
idlocs..............................................................85
if.................................................................86
else..............................................................68
end.................................................................70
ifdef.............................................................88
INCLUDE/FI in Linker Scripts............................187
ifndef............................................................89
include..........................................................90
Index
Assembler/Linker/Librarian User’s Guide

Include File ...................................................... 28
Increment ....................................................... 41
Initialized Data .................................................. 194

Input/Output Files
Assembler .......................................................... 26
Librarian ............................................................ 237
Linker ............................................................... 173

Instruction Operands .............................................. 142
Instruction Sets .................................................. 251
12-Bit Core ...................................................... 253, 259
12-Bit/14-Bit Cores ........................................... 259
14-Bit Core ....................................................... 254
PIC18 Device ..................................................... 260

Internet Address, Microchip .................................. 6

Interrupt Handling
PIC16 Example .................................................. 75, 81, 99, 124, 129
PIC18 Example .................................................. 81, 101

L
Labels .................................................................. 27
LIBPATH ........................................................... 182
Library File ...................................................... 12, 174, 237
Library Path ...................................................... 196, 197

Limitations
Assembler .......................................................... 167
Linker Processing ............................................... 191

Linker Scripts ...................................................... 12, 174, 181
Debug Tool ....................................................... 181
Standard ............................................................ 181

list ..................................................................... 91

Listing Directives ................................................. 45
error .................................................................. 72
errorlevel ......................................................... 73
list .................................................................... 91
messg .............................................................. 96
nolist ............................................................... 98
page ............................................................... 101
space .............................................................. 109
subtitle ............................................................ 109
title .................................................................. 110

Listing File ........................................................ 28, 45, 174
LKRPATH ........................................................... 182
local ............................................................... 92
Logical Sections .................................................. 186
low .................................................................. 40, 143

M
Macro
Code Examples .................................................. 153
End .................................................................. 70
Exit .................................................................. 75
Expand ............................................................ 77
No Expansion .................................................. 98
Text Substitution .............................................. 152
Usage .............................................................. 152

macro ............................................................ 94

Macro Directives .................................................. 45
Defined ........................................................... 152
dendm ............................................................. 70
dexit ............................................................... 75
dexpand .......................................................... 77

local ............................................................ 92
macro ........................................................... 94
noexpand ...................................................... 98
Macro Language ............................................... 151
Macro Syntax .................................................... 151
Macros ........................................................... 27
Macros in Linker Scripts .................................. 187
main .............................................................. 39
Map File ......................................................... 176, 196, 197
Maximum RAM Location .................................. 95
Maximum ROM Location .................................. 96
maxram .......................................................... 95
maxrom .......................................................... 96
MCC_INCLUDE .............................................. 198, 199
Message .......................................................... 96
Messages
Assembler ....................................................... 165
messg ............................................................. 96
Mnemonics ....................................................... 27
mp2cod Utility .................................................. 246
mp2hex Utility .................................................. 246
MPASM Assembler .......................................... 13
MPASM Assembler Overview .......................... 23
mpasm.exe ..................................................... 167
mpasmwin.exe .............................................. 9, 23, 33
MPLAB C18 C Compiler .................................. 12, 15
MPLAB IDE Build Options Dialog
MPASM Assembler Tab ....................................... 13
MPASM/C17/C18 Suite Tab ............................... 16
MPLAB C17 Tab ............................................... 14
MPLAB C18 Tab ............................................... 15
MPLINK LinkerTab ............................................ 16
MPLIB Librarian Overview ............................... 235
MPLIB Object Librarian ................................. 12, 16
mplib.exe ...................................................... 9
MPLINK Linker Overview ................................. 171
MPLINK Object Linker ................................. 12, 16
mplink.exe ................................................... 9

noexpand ....................................................... 98
nolist ............................................................. 98
NOT, logical .................................................. 40

O
Object File ........................................................ 32, 174, 237
Object File Directives ...................................... 45
access_ovr ..................................................... 46
bankisel ......................................................... 48
banksel .......................................................... 50
code ............................................................ 54
code_pack ..................................................... 55
eextern .......................................................... 78
global ............................................................ 82
idata .............................................................. 82
idata_acs ....................................................... 84
pagesel ......................................................... 102

2009 Microchip Technology Inc.
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pageselw ................................................................. 103</td>
<td>Symbol Constant .......................................................... 58</td>
</tr>
<tr>
<td>udata ........................................................................ 110</td>
<td>Symbols, In Expressions ............................................... 40</td>
</tr>
<tr>
<td>udata_acs .................................................................. 111</td>
<td></td>
</tr>
<tr>
<td>udata_ovr .................................................................. 113</td>
<td>T</td>
</tr>
<tr>
<td>udata_shr .................................................................. 115</td>
<td></td>
</tr>
<tr>
<td>Object Files, Precompiled ........................................ 12</td>
<td>Table, Define .............................................................. 67</td>
</tr>
<tr>
<td>Object Module, Generating ........................................ 148</td>
<td>Templates ................................................................. 200</td>
</tr>
<tr>
<td>Operands ........................................................................ 28</td>
<td>Text Strings .................................................................. 200</td>
</tr>
<tr>
<td>Operators, Arithmetic .............................................. 40</td>
<td>Text Substitution Label.................................................. 65</td>
</tr>
<tr>
<td>OR, logical ................................................................... 41</td>
<td>Tips and Tricks</td>
</tr>
<tr>
<td>org ............................................................................ 99</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Bit Shifting Using Carry Bit .................................... 139</td>
</tr>
<tr>
<td>page ............................................................................. 101</td>
<td>Conditional Bit Set/Clear ............................................. 138</td>
</tr>
<tr>
<td>Page Eject .................................................................... 101</td>
<td>Delay Techniques ......................................................... 137</td>
</tr>
<tr>
<td>Page Selection .......................................................... 102</td>
<td>Optimizing Destinations .............................................. 138</td>
</tr>
<tr>
<td>Page Selection - WREG .............................................. 103</td>
<td>Swap File Register with W ............................................ 139</td>
</tr>
<tr>
<td>pagesel ....................................................................... 102, 147</td>
<td>Using External Memory ................................................ 140</td>
</tr>
<tr>
<td>pageselw .................................................................. 103</td>
<td></td>
</tr>
<tr>
<td>Paging ........................................................................ 127, 147</td>
<td></td>
</tr>
<tr>
<td>PATH .......................................................................... 198, 199</td>
<td>title ............................................................................. 110</td>
</tr>
<tr>
<td>Processing, Linker ..................................................... 191</td>
<td></td>
</tr>
<tr>
<td>processor .................................................................... 104</td>
<td></td>
</tr>
<tr>
<td>Processor, Set ................................................................ 91, 104, 123</td>
<td></td>
</tr>
<tr>
<td>Program Memory .......................................................... 142</td>
<td></td>
</tr>
<tr>
<td>Projects ........................................................................ 11</td>
<td></td>
</tr>
<tr>
<td>PROTECTED .................................................................. 183</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>U</td>
</tr>
<tr>
<td>Radix ........................................................................... 39</td>
<td>udata ............................................................................. 110, 145</td>
</tr>
<tr>
<td>radix ............................................................................ 105</td>
<td>udata_acs ........................................................................ 111, 145</td>
</tr>
<tr>
<td>Radix, Set ..................................................................... 91, 105, 123</td>
<td>udata_ovr ......................................................................... 113, 145</td>
</tr>
<tr>
<td>RAM Allocation ............................................................. 145</td>
<td>udata_shr ......................................................................... 115, 145</td>
</tr>
<tr>
<td>RAM Memory Regions, Defining .................................... 183</td>
<td>undefine .......................................................................... 116</td>
</tr>
<tr>
<td>Reading, Recommended .................................................. 5</td>
<td>Unimplemented RAM ..................................................... 46</td>
</tr>
<tr>
<td>Register Assignments .................................................... 127</td>
<td>Unimplemented ROM .................................................... 47</td>
</tr>
<tr>
<td>relocatable .................................................................... 32</td>
<td>upper ........................................................................... 40, 143</td>
</tr>
<tr>
<td>Relocatable Code, Generating ........................................ 25</td>
<td></td>
</tr>
<tr>
<td>Relocatable Objects ......................................................... 141</td>
<td>Utilities Overview and Usage ......................................... 245</td>
</tr>
<tr>
<td>res ................................................................................ 106</td>
<td></td>
</tr>
<tr>
<td>Reserved Section Names, Assembler .............................. 39</td>
<td>V</td>
</tr>
<tr>
<td>Reserved Words, Assembler ........................................... 39</td>
<td>Variable</td>
</tr>
<tr>
<td>ROM Memory Regions, Defining ...................................... 185</td>
<td>Declare .......................................................................... 117</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>Sample Applications, Linker ......................................... 195</td>
<td></td>
</tr>
<tr>
<td>Scripts, Linker ............................................................. 181</td>
<td></td>
</tr>
<tr>
<td>Search Order, Include Files ......................................... 90</td>
<td></td>
</tr>
<tr>
<td>SECTION ................................................................. 183, 186</td>
<td></td>
</tr>
<tr>
<td>set ................................................................................. 108</td>
<td></td>
</tr>
<tr>
<td>Set Program Origin ........................................................ 99</td>
<td></td>
</tr>
<tr>
<td>SHAREBANK ............................................................. 183, 186</td>
<td></td>
</tr>
<tr>
<td>Simple .......................................................................... 47, 48</td>
<td></td>
</tr>
<tr>
<td>Source Code ................................................................. 12</td>
<td></td>
</tr>
<tr>
<td>Source Code File, Assembly .......................................... 26</td>
<td></td>
</tr>
<tr>
<td>space ........................................................................... 109</td>
<td></td>
</tr>
<tr>
<td>Stack ........................................................................... 186</td>
<td></td>
</tr>
<tr>
<td>STACK SIZE .............................................................. 186</td>
<td></td>
</tr>
<tr>
<td>Standard Linker Scripts .............................................. 181</td>
<td></td>
</tr>
<tr>
<td>Store Strings in Program Memory .................................. 59</td>
<td></td>
</tr>
<tr>
<td>subtitle ........................................................................ 109</td>
<td></td>
</tr>
</tbody>
</table>

© 2009 Microchip Technology Inc. DS33014K-page 287
### WORLDWIDE SALES AND SERVICE

#### AMERICAS

**Corporate Office**  
2355 West Chandler Blvd.  
Chandler, AZ  85224-6199  
Tel:  480-792-7200  
Fax:  480-792-7277  
Technical Support: http://support.microchip.com  
Web Address: www.microchip.com

- **Atlanta**  
  Duluth, GA  
  Tel: 678-957-9614  
  Fax: 678-957-1455
- **Boston**  
  Westborough, MA  
  Tel: 774-760-0087  
  Fax: 774-760-0088
- **Chicago**  
  Itasca, IL  
  Tel: 630-285-0071  
  Fax: 630-285-0075
- **Cleveland**  
  Independence, OH  
  Tel: 216-447-0064  
  Fax: 216-447-0063
- **Dallas**  
  Addison, TX  
  Tel: 972-818-7423  
  Fax: 972-818-2924
- **Detroit**  
  Farmington Hills, MI  
  Tel: 248-538-2250  
  Fax: 248-538-2260
- **Kokomo**  
  Kokomo, IN  
  Tel: 765-864-8360  
  Fax: 765-864-8387
- **Los Angeles**  
  Mission Viejo, CA  
  Tel: 949-462-9523  
  Fax: 949-462-9608
- **Santa Clara**  
  Santa Clara, CA  
  Tel: 408-961-6444  
  Fax: 408-961-6445
- **Toronto**  
  Mississauga, Ontario, Canada  
  Tel: 905-673-0699  
  Fax: 905-673-6509

#### ASIA/PACIFIC

- **Asia Pacific Office**  
  Suites 3707-14, 37th Floor  
  Tower 6, The Gateway  
  Harbour City, Kowloon  
  Hong Kong  
  Tel: 852-2401-1200  
  Fax: 852-2401-3431
- **Australia - Sydney**  
  Tel: 61-2-9868-6733  
  Fax: 61-2-9868-6755
- **China - Beijing**  
  Tel: 86-10-8528-2100  
  Fax: 86-10-8528-2104
- **China - Chengdu**  
  Tel: 86-28-8665-5511  
  Fax: 86-28-8665-7889
- **China - Hong Kong SAR**  
  Tel: 852-2401-1200  
  Fax: 852-2401-3431
- **China - Nanjing**  
  Tel: 86-25-8473-2460  
  Fax: 86-25-8473-2470
- **China - Qingdao**  
  Tel: 86-532-8502-7355  
  Fax: 86-532-8502-7205
- **China - Shanghai**  
  Tel: 86-21-5407-5066  
  Fax: 86-21-5407-5066
- **China - Shenzhen**  
  Tel: 86-755-8203-2660  
  Fax: 86-755-8203-2660
- **China - Zhuhai**  
  Tel: 86-756-3210040  
  Fax: 86-756-3210049

#### ASIA/PACIFIC

- **India - Bangalore**  
  Tel: 91-80-3090-4444  
  Fax: 91-80-3090-4080
- **India - New Delhi**  
  Tel: 91-11-4160-8631  
  Fax: 91-11-4160-8632
- **India - Pune**  
  Tel: 91-20-2566-1512  
  Fax: 91-20-2566-1513
- **Japan - Yokohama**  
  Tel: 81-45-471-6166  
  Fax: 81-45-471-6122
- **Korea - Daegu**  
  Tel: 82-53-744-4301  
  Fax: 82-53-744-4302
- **Korea - Seoul**  
  Tel: 82-2-554-7200  
  Fax: 82-2-558-5932 or 82-2-558-5934
- **Malaysia - Kuala Lumpur**  
  Tel: 60-3-6201-9857  
  Fax: 60-3-6201-9859
- **Malaysia - Penang**  
  Tel: 60-4-227-8870  
  Fax: 60-4-227-4068
- **Philippines - Manila**  
  Tel: 63-2-634-9065  
  Fax: 63-2-634-9069
- **Singapore**  
  Tel: 65-6334-8870  
  Fax: 65-6334-8850
- **Taiwan - Hsin Chu**  
  Tel: 886-3-6578-300  
  Fax: 886-3-6578-370
- **Taiwan - Kaohsiung**  
  Tel: 886-7-536-4818  
  Fax: 886-7-536-4803
- **Taiwan - Taipei**  
  Tel: 886-2-2508-0102  
  Fax: 886-2-2508-0102
- **Thailand - Bangkok**  
  Tel: 66-2-694-1351  
  Fax: 66-2-694-1350

#### EUROPE

- **Austria - Wels**  
  Tel: 43-7242-2244-39  
  Fax: 43-7242-2244-393
- **Denmark - Copenhagen**  
  Tel: 45-4450-2828  
  Fax: 45-4485-2829
- **France - Paris**  
  Tel: 33-1-69-53-63-20  
  Fax: 33-1-69-30-90-79
- **Germany - Munich**  
  Tel: 49-89-627-144-0  
  Fax: 49-89-627-144-44
- **Italy - Milan**  
  Tel: 39-0331-742611  
  Fax: 39-0331-466781
- **Netherlands - Drunen**  
  Tel: 31-646-690399  
  Fax: 31-646-690340
- **Spain - Madrid**  
  Tel: 34-91-708-08-90  
  Fax: 34-91-708-08-91
- **UK - Wokingham**  
  Tel: 44-118-921-5869  
  Fax: 44-118-921-5820