MPLAB® XC32 ASSEMBLER, LINKER AND UTILITIES
User's Guide

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

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

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

Trademarks

Printed on recycled paper.

ISBN: 978-1-62077-521-9

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949

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

MPLAB® XC32 ASSEMBLER, LINKER AND UTILITIES USER'S GUIDE

Table of Contents

Preface

NOTICE TO CUSTOMERS

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

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

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

INTRODUCTION

This chapter contains general information that is useful to know before using 32-bit language tools. Items discussed include:

- Document Layout
- Conventions Used in this Guide
- Recommended Reading
- The Microchip Web Site
- myMicrochip Personalized Notification Service
- Customer Support

DOCUMENT LAYOUT

This document describes how to use GNU language tools to write code for 32-bit applications. The document layout is as follows:

Part 1 – MPLAB XC32 Assembler

- **Chapter 1. Assembler Overview** – gives an overview of assembler operation.
- **Chapter 2. Assembler Command-Line Options** – details command-line options for the assembler.
- **Chapter 3. MPLAB XC32 Assembly Language** – discusses the source language used by the macro assembler.
- **Chapter 4. Assembler Directives** – describes the assembler commands in the source code.
- **Chapter 5. Assembler Errors/Warnings/Messages** – provides a descriptive list of the errors, warnings and messages.

Part 2 – MPLAB XC32 Object Linker

- **Chapter 6. Linker Overview** – gives an overview of linker operation.
- **Chapter 7. Linker Command-Line Interface** – details command-line options for the linker.
- **Chapter 8. Linker Scripts** – describes how to generate and use linker scripts to control linker operation.
- **Chapter 9. Linker Processing** – discusses how the linker builds an application from input files.
- **Chapter 10. Linker Examples** – includes a number of 32-bit-specific linker examples.
- **Chapter 11. Linker Errors/Warnings** – provides a descriptive list of the errors and warnings.

Part 3 – 32-Bit Utilities (including the Archiver/Librarian)

- **Chapter 12. MPLAB XC32 Object Archiver/Librarian** – details command-line options for the archiver/librarian.
- **Chapter 13. Other Utilities** – details the other utilities and their operation.

Part 4 – Appendices

- **Appendix A. Deprecated Features** – discusses the features considered obsolete.
- **Appendix B. Useful Tables** – lists some useful tables: the ASCII character set and hexadecimal to decimal conversion.
- **Appendix C. GNU Free Documentation License** – details the license requirements for using the GNU language tools.

CONVENTIONS USED IN THIS GUIDE

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

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

RECOMMENDED READING

This documentation describes how to use 32-bit language tools. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Readme Files

For the latest information on Microchip tools, read the associated Readme files (HTML files) included with the software.

MPLAB® XC32 C/C++ Compiler User's Guide (DS51686)

A guide to using the 32-bit C compiler. The 32-bit linker is used with this tool.

32-Bit Language Tools Libraries (DS51685)

A descriptive listing of libraries available for Microchip 32-bit devices. This includes standard (including math) libraries and compiler built-in functions. 32-bit peripheral libraries are described in HTML files provided with each peripheral library type.

Device-Specific Documentation

The Microchip website contains many documents that describe 32-bit device functions and features. Among these are:

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

THE MICROCHIP WEB SITE

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

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

myMICROCHIP PERSONALIZED NOTIFICATION SERVICE

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

Please visit http://www.microchip.com/pcn to begin the registration process and select your preferences to receive personalized notifications. A FAQ and registration details are available on the page, which can be opened by selecting the link above.

When you are selecting your preferences, choosing "Development Systems" will populate the list with available development tools. The main categories of tools are listed below:

- **Compilers** – The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on the Microchip MPLAB REAL ICE™ In-Circuit Emulator.
- **In-Circuit Debuggers** – The latest information on Microchip MPLAB ICD 3 In-Circuit Debugger and PICkit™ 3 In-Circuit Debugger/Programmer.
- **MPLAB® X IDE** – The latest information on Microchip MPLAB X IDE, the Integrated Development Environment for development systems tools. This list is focused on the MPLAB X IDE, MPLAB X IDE Project Manager, MPLAB X Editor and MPLAB X SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the device (production) programmers MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 In-Circuit Debugger, MPLAB PM3 and development (non-production) programmers PICkit 3 In-Circuit Debugger/Programmer.
- **Starter/Demo Boards** – These include MPLAB Starter Kit boards, PICDEM™ demonstration boards, and various other evaluation boards.

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.

Documentation errors or comments may be emailed to docerrors@microchip.com.

Part 1 – MPLAB XC32 Assembler

NOTES:

Chapter 1. Assembler Overview

1.1 INTRODUCTION

MPLAB® XC32 Assembler (xc32-as) produces relocatable machine code from symbolic assembly language for the PIC32 MCU family of devices. The assembler is a Windows® operating system console application that provides a platform for developing assembly language code. The assembler is a port of the GNU assembler from the Free Software Foundation.

Topics covered in this chapter are:

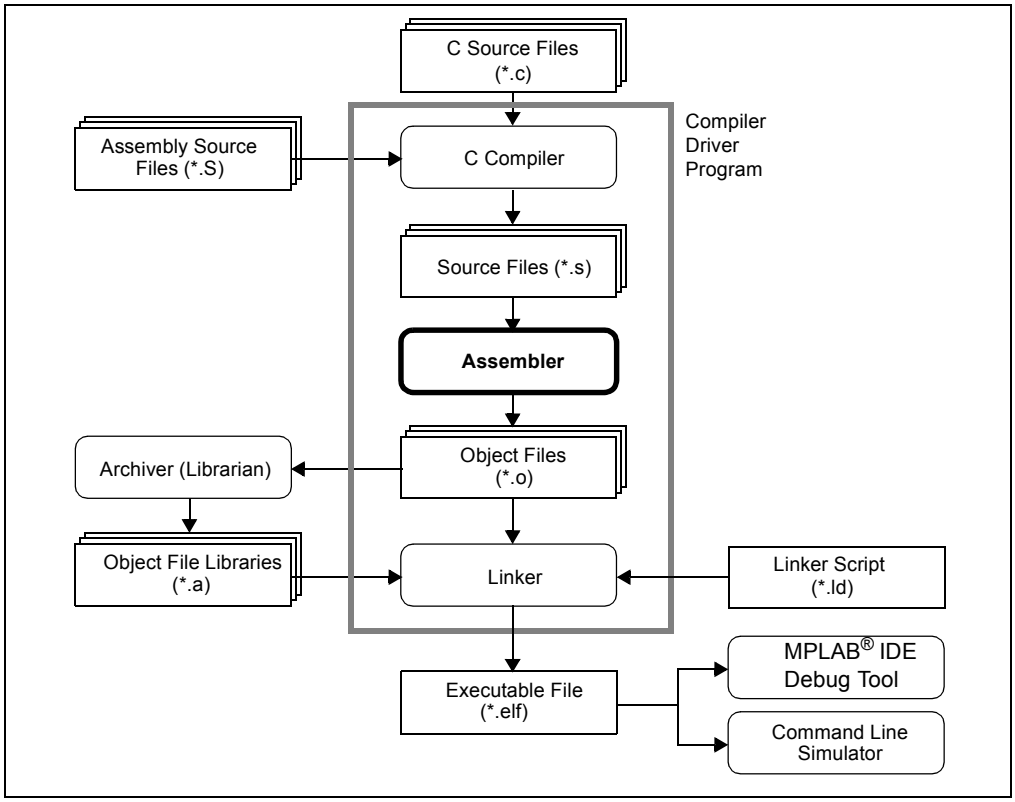
- Assembler and Other Development Tools
- Feature Set
- Input/Output Files

1.2 ASSEMBLER AND OTHER DEVELOPMENT TOOLS

MPLAB XC32 Assembler translates user assembly source files. In addition, the MPLAB XC32 C/C++ Compiler uses the assembler to produce its object file.

After the C preprocessor processes the assembly source file (*.S), the assembler generates a relocatable object file that can then be put into an archive or linked with other relocatable object files and archives to create an executable file. See Figure 1-1 for an overview of the tools process flow.

FIGURE 1-1: TOOLS PROCESS FLOW



1.3 FEATURE SET

Notable features of the assembler include:

- Support for the MIPS32, MIPS16e and microMIPS instruction sets
- Support for ELF object format
- Available for Linux®, OS X®, and and Windows operating systems
- Rich Directive Set
- Flexible Macro Language
- Command-Line Interface
- Integrated component of MPLAB X IDE

1.4 INPUT/OUTPUT FILES

Standard assembler input and output files are listed below.

Extension	Description
Input	
.S	Assembly source file to be preprocessed (recommended)
.s	Source File
Output	
.o	Object File
.1st	Listing File

Unlike the MPASM™ Assembler (for use with 8-bit PIC® MCUs), MPLAB XC32 Assembler does not generate error files, hex files, or symbol and debug files. The XC32 assembler is capable of creating a listing file and a relocatable object file (that may or may not contain debugging information). MPLAB Linker for PIC32 MCUs is used with the assembler to produce the final object files, map files and a final executable file for debugging with MPLAB X IDE (see Figure 1-1).

1.4.1 Source Files

The assembler accepts, as input, a source file that consists of PIC32 instructions, assembler directives and comments. A sample source file is shown in Example 1-1.

Note: Microchip Technology strongly suggests an .S extension for assembly source files. This will enable you to easily use the C compiler driver without having to specify the option to tell the driver that the file should be treated as an assembly file. The capitalized S also indicates that the source file should be preprocessed by the C preprocessor before being passed to the assembler. See the "*MPLAB XC32 C/C++ Compiler User's Guide*" (DS51686) for more details on the C compiler driver.

EXAMPLE 1-1: SAMPLE ASSEMBLER CODE

```
Updated example code:

#include <xc.h>
#define IOPORT_BIT_7 (1 << 7)
    .global main    /* define all global symbols here */
    .text
    /* define which section (for example "text")
     * does this portion of code resides in. Typically,
     * all your code will reside in .text section as
     * shown below.
     */
    .set noreorder
    /* This is important for an assembly programmer. This
     * directive tells the assembler not to optimize
     * the order of the instructions as well as not to insert
     * 'nop' instructions after jumps and branches.
     */
/*****************************************************************
     * main()
     * This is where the PIC32 start-up code will jump to after initial
     * set-up.
     *****************************************************************/
```

```
   .ent main /* directive that marks symbol 'main' as function in the ELF
             * output
             */
main:
   /* Call function to clear bit relevant to pin 7 of port A.
    * The 'jal' instruction places the return address in the $ra
    * register.
    */
   ori      a0, $0, IOPORT_BIT_7
   jal      mPORTAClearBits
   nop
   /* endless loop */
endless:
   j endless
   nop
   .end main /* directive that marks end of 'main' function and its
             * size in the ELF output
             */
/********************************************************************
 * mPORTAClearBits(int bits)
 * This function clears the specified bits of IOPORT A.
 *
 * pre-condition: $ra contains return address
 * Input: Bit mask in $a0
 * Output: none
 * Side effect: clears bits in IOPORT A
 ********************************************************************/
   .ent mPORTAClearBits
mPORTAClearBits:
   /* function prologue - save registers used in this function
    * on stack and adjust stack-pointer
    */
   addiu   sp, sp, -4
   sw      s0, 0(sp)
   la      s0, LATACLR
   sw      a0, 0(s0)      /* clear specified bits */
   /* function epilogue - restore registers used in this function
    * from stack and adjust stack-pointer
    */
   lw      s0, 0(sp)
   addiu   sp, sp, 4
   /* return to caller */
   jr      ra
   nop
   .end mPORTAClearBits
```

1.4.2 Object File

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

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

1.4.3 Listing File

The assembler has the capability to produce a listing file. The listing file is not an absolute listing file, and the addresses that appear in the listing are relative to the start of its section.

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

A listing file produced by the assembler is composed of the elements listed below. Example 1-2 shows a sample listing file.

1.4.3.1 HEADER

The header contains the name of the assembler, the name of the file being assembled, and a page number. This is not shown if the `-an` option is specified.

1.4.3.2 TITLE

The title line contains the title specified by the `.title` directive. This is not shown if the -an option is specified.

1.4.3.3 SUBTITLE

The subtitle line contains the subtitle specified by the `.sbttl` directive. This is not shown if the `-an` option is specified.

1.4.3.4 HIGH-LEVEL SOURCE

High-level source will be present if the `-ah` option is given to the assembler. The format for high-level source is:

```
<line #>:<filename>       **** <source>
```

For example:

```
1:hello.c       **** #include <stdio.h>
```

1.4.3.5 ASSEMBLER SOURCE

Assembler source will be present if the `-al` option is given to the assembler. The format for assembler source is:

```
<line #> <addr> <encoded bytes> <source>
```

For example:

```
35 0000 80000434    ori      $a0, $zero, IOPORT_BIT_7
```

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

1.4.3.6 SYMBOL TABLE

A symbol table is present if the `-as` option is given to the assembler. A list of defined symbols and a list of undefined symbols will be given.

The defined symbols will have the format:

```
DEFINED SYMBOLS
<filename>:<line #> <section>:<addr> <symbol>
```

For example:

```
DEFINED SYMBOLS
foo.S:79     .text:00000000 main
foo.S:107    .text:00000014 mPORTAClearBits
```

The undefined symbols will have the format:

```
UNDEFINED SYMBOLS
<symbol>
```

For example:

```
UNDEFINED SYMBOLS
WDTCON
WDTCONCLR
```

EXAMPLE 1-2: SAMPLE ASSEMBLER LISTING FILE

```
GAS LISTING foo.s      page 1

   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                 .nolist
   0
   0
   3                 .list
   4
   5              #define IOPORT_BIT_7 (1 << 7)
   6
   8                 .global baz /* define all global symbols here */
   9
  10                 /* define which section (for example "text")
  11                  * does this portion of code resides in.
  12                  * Typically, all of your code will reside in
                       * the .text section as shown.
  14                  */
  15                 .text
  16
  17                 /* This is important for an assembly programmer.
  18                  * This directive tells the assembler not to
  19                  * optimize the order of the instructions
  20                  * as well as not to insert 'nop' instructions
  21                  * after jumps and branches.
  22                  */
  23                 .set noreorder
  24
  25              .ent baz /* directive that marks symbol 'baz' as
  26                        * a function in ELF output
  27                        */
  28
  29              baz:
  30
  31                 /* Call function to clear bit relevant to pin
  32                  * 7 in port A. The 'jal' instruction places
```

```
  33                  * the return address in $ra.
  34                  */
  35 0000 80000434     ori      $a0, $zero, IOPORT_BIT_7
  36 0004 0500000C     jal      mPORTAClearBits
  37 0008 00000000     nop
  38
  39                 /* endless loop */
  40              endless:
  41 000c 03000008     j      endless
  42 0010 00000000     nop
  43
  44               .end baz /* directive that marks end of 'baz'
  45                        * function and registers size in ELF
  46                        * output
  47                        */
DEFINED SYMBOLS
                           *ABS*:00000000 foo.S
                           *ABS*:00000001 __DEBUG
               foo.S:56    .text:00000014 mPORTAClearBits
               foo.S:38    .text:0000000c endless
```

NOTES:

Chapter 2. Assembler Command-Line Options

2.1 INTRODUCTION

MPLAB XC32 Assembler (xc32-as) may be used on the host PC's command-line interface (e.g., cmd.exe) as well as with the MPLAB X IDE project manager.

The MPLAB X IDE project manager automatically calls the assembler (via the xc32-gcc compilation driver) when building a project. Many of the commonly used options listed here are available as check boxes on the MPLAB X IDE project build-options dialog. However, for a more advanced option, you may have to specify the option in the "Alternate Settings" field of the dialog. After you build a project in MPLAB X IDE, the output window shows the options passed to the assembler.

Topics covered in this chapter are:

- Assembler Interface Syntax
- Compilation-Driver Interface Syntax
- Options that Modify the Listing Output
- Options that Control Informational Output
- Options that Control Output File Creation
- Assembler Symbol-Definition and Search-Path Options
- Compilation-Driver and Preprocessor Options

2.2 ASSEMBLER INTERFACE SYNTAX

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

```
xc32-as [options|sourcefiles]...
```

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

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

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

Note: command-line options are case sensitive.

2.3 COMPILATION-DRIVER INTERFACE SYNTAX

The compilation-driver program (xc32-gcc) preprocesses, compiles, assembles, and links C and assembly-language modules and library archives. This driver orchestrates the build process so that you often don't need to know which individual programs preprocess, compile, assemble, and link. The driver calls the appropriate individual tools to complete the requested build process.

In practice, the assembler is usually invoked via xc32-gcc, which determines that it should assemble an input file by its `*.S` or `*.s` filename extension. The compilation driver sends a file with a `*.S` (upper case) extension through the CPP-style preprocessor before it passes the file to the assembler while the driver sends a file with a `*.s` (lower case) extension directly to the assembler.

The basic form of the compilation-driver command line is:

```
xc32-gcc [options] files
```

> **Note:** Command-line options and filename extensions are case sensitive.

To pass an assembler option from the compilation driver to the assembler, use the `-Wa`,option option. The option argument should not contain white space.

EXAMPLE 2-1: EXAMPLE COMPILATION-DRIVER COMMAND LINE

```
xc32-gcc -mprocessor=32MX360F512L -I"./include" ASMfile.S
  -o"ASMfile.o" -DMYMACRO=1 -Wa,-ah="ASMfile.lst"
```

For additional information on the compilation driver, see the "*MPLAB XC32 C/C++ Compiler User's Guide*" (DS51686).

> **Note:** To use the xc32-gcc compilation driver from MPLAB X IDE, be sure to select the XC32 Compiler Toolchain for your project.

2.4 OPTIONS THAT MODIFY THE LISTING OUTPUT

The following options are used to control the listing output. A listing file is helpful for debugging and general analysis of code operation. Use the following options to construct a listing file with information that you find useful.

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

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

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

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

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

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

2.4.1.1 -ac

-ac omits false conditionals from a listing. Any lines that are not assembled because of a false .if or .ifdef (or the .else of a true .if or .ifdef) will be omitted from the listing. Example 2-2 shows a listing where the -ac option was not used. Example 2-3 shows a listing for the same source where the -ac option was used.

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

```
GAS LISTING asm.s                       page 1

   1              # 1 "asm.S"
   2              # 1 "<built-in>"
   1                    .data
   0
   2                    .if 0
   3                      .if 1
   4                      .endif
   5                      .long 0
   6                      .if 0
   7                        .long 0
   8                      .endif
   9                    .else
  10                      .if 1
  11                      .endif
  12 0000 02000000         .long 2
  13                      .if 0
  14                        .long 3
  15                      .else
  16 0004 04000000           .long 4
  17                      .endif
  18                    .endif
  19                    .if 0
  20                      .long 5
  21                    .elseif 1
  22                      .if 0
  23                        .long 6
  24                      .elseif 1
  25 0008 07000000           .long 7
  26                      .endif
  27                    .elseif 1
  28                      .long 8
  29                    .else
  30                      .long 9
  31                    .endif
```

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

```
GAS LISTING asm.s page 1

   1              # 1 "asm.S"
   2              # 1 "<built-in>"
   1                    .data
   0
   0
   2                   .if 0
   9                   .else
  10                     .if 1
  11                     .endif
  12 0000 02000000         .long 2
  13                     .if 0
  15                     .else
  16 0004 04000000           .long 4
  17                     .endif
  18                  .endif
  19                  .if 0
  21                  .elseif 1
  22                     .if 0
  24                     .elseif 1
  25 0008 07000000           .long 7
  26                     .endif
  27                  .elseif 1
  29                  .else
  31                  .endif
```

Note: Some lines omitted due to `-ac` option.

2.4.1.2 -ad

-ad omits debugging directives from the listing. This option is useful when processing compiler-generated assembly code containing debugging information, the compiler-generated debugging directives will not clutter the listing. Example 2-4 shows a listing using both the d and h sub-options. Compared to using the h sub-option alone (see the next section), the listing is much cleaner.

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

```
GAS LISTING test.s page 1


   1              .section .mdebug.abi32
   2              .previous
  10              .Ltext0:
  11              .align2
  12              .globlmain
  13              .LFB0:
  14              .file 1 "src\\test.c"
   1:src/test.c    **** #include <xc.h>
   2:src/test.c    **** volatile unsigned int testval;
   3:src/test.c    ****
   4:src/test.c    **** int
   5:src/test.c    **** main (void)
   6:src/test.c    **** {
  15              .loc 1 6 0
  16              .setnomips16
  17              .entmain
  18             main:
  19         .frame$fp,8,$31# vars= 0, regs= 1/0, args= 0, gp= 0
  20         .mask0x40000000,-8
  21         .fmask0x00000000,0
  22              .setnoreorder
  23              .setnomacro
  24
  25 0000 F8FFBD27 addiu$sp,$sp,-8
  26              .LCFI0:
  27 0004 0000BEAF sw$fp,0($sp)
  28              .LCFI1:
  29 0008 21F0A003 move$fp,$sp
  30              .LCFI2:
   7:src/test.c    ****   testval += 1;
  31              .loc 1 7 0
  32 000c 0000828F lw$2,%gp_rel(testval)($28)
  33 0010 01004224 addiu$2,$2,1
  34 0014 000082AF sw$2,%gp_rel(testval)($28)
   8:src/test.c    ****   return 0;
  35              .loc 1 8 0
  36 0018 21100000 move$2,$0
   9:src/test.c    **** }
  37              .loc 1 9 0
  38 001c 21E8C003 move$sp,$fp
  39 0020 0000BE8F lw$fp,0($sp)
  40 0024 0800BD27 addiu$sp,$sp,8
  41 0028 0800E003 j$31
  42 002c 00000000 nop
```

```
43
44              .setmacro
45              .setreorder
46              .endmain
47              .LFE0:
49
50              .commtestval,4,4
88              .Letext0:
```

2.4.1.3 -ah

-ah requests a high-level language listing. High-level listings require that the assembly source code was generated by a compiler, a debugging option like -g was given to the compiler, and that assembly listings (-al) are also requested. -al requests an output program assembly listing. Example 2-5 shows a listing that was generated using the -alh command line option.

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

```
GAS LISTING tempfile.s page 1


   1              .section .mdebug.abi32
   2              .previous
   3              .section.debug_abbrev,"",@progbits
   4              .Ldebug_abbrev0:
   5              .section.debug_info,"",@progbits
   6              .Ldebug_info0:
   7              .section.debug_line,"",@progbits
   8              .Ldebug_line0:
   9 0000 34000000 .text
  11              .align2
  12              .globlmain
  13              .LFB0:
  14              .file 1 "src/test.c"
   1:src/test.c    **** #include <xc.h>
   2:src/test.c    **** volatile unsigned int testval;
   3:src/test.c    ****
   4:src/test.c    **** int
   5:src/test.c    **** main (void)
   6:src/test.c    **** {
  15              .loc 1 6 0
  16              .setnomips16
  17              .entmain
  18             main:
  19        .frame$sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0
  20              .mask0x00000000,0
  21              .fmask0x00000000,0
  22              .setnoreorder
  23              .setnomacro
  24
   7:src/test.c    ****   testval += 1;
  25              .loc 1 7 0
  26 0000 0000848F lw$4,%gp_rel(testval)($28)
   8:src/test.c    ****   return 0;
   9:src/test.c    **** }
  27              .loc 1 9 0
  28 0004 21100000 move$2,$0
  29              .loc 1 7 0
  30 0008 01008324 addiu$3,$4,1
  31 000c 000083AF sw$3,%gp_rel(testval)($28)
  32              .loc 1 9 0
  33 0010 0800E003 j$31
  34 0014 00000000 nop
```

```
35
36              .setmacro
37              .setreorder
38              .endmain
39              .LFE0:
40              .sizemain, .-main
41
42              .commtestval,4,4
```

2.4.1.4 -al

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

2.4.1.5 -am

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

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

```
GAS LISTING foo.s                       page 1


   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                      .macro  sum from=0, to=5
   0
   0
   2                      .long   \from
   3                      .if     \to-\from
   4                      sum     "(\from+1)",\to
   5                      .endif
   6                      .endm
   7
   8                      .data
   9 0000 00000000         .long 0
  10 0004 0A000000         sum 10, 14
  10      0B000000
  10      0C000000
  10      0D000000
  10      0E000000
  11 0018 00000000         .long 0
```

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

```
GAS LISTING foo.s                       page 1


   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                      .macro  sum from=0, to=5
   0
   0
   2                      .long   \from
   3                      .if     \to-\from
   4                      sum     "(\from+1)",\to
   5                      .endif
   6                      .endm
   7
   8                      .data
   9 0000 00000000         .long 0
  10                      sum 10, 14
  10 0004 0A000000 > .long 10
  10              > .if 14-10
  10              > sum "(10+1)",14
  10 0008 0B000000 >> .long (10+1)
  10              >> .if 14-(10+1)
  10              >> sum "((10+1)+1)",14
  10 000c 0C000000 >>> .long ((10+1)+1)
  10              >>> .if 14-((10+1)+1)
  10              >>> sum "(((10+1)+1)+1)",14
  10 0010 0D000000 >>>> .long (((10+1)+1)+1)
  10              >>>> .if 14-(((10+1)+1)+1)
  10              >>>> sum "((((10+1)+1)+1)+1)",14
  10 0014 0E000000 >>>>> .long ((((10+1)+1)+1)+1)
  10              >>>>> .if 14-((((10+1)+1)+1)+1)
  10              >>>>> sum "(((((10+1)+1)+1)+1)+1)",14
  10              >>>>> .endif
  10              >>>> .endif
  10              >>> .endif
  10              >> .endif
  10              > .endif
  11 0018 00000000         .long 0
```

Note: > signifies expanded macro instructions.

2.4.1.6 -an

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

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

```
GAS LISTING foo.s                       page 1
User's Guide Example
Listing Options
   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                 .text
   0
   0
   2                 .title "User's Guide Example"
   3                 .sbttl "Listing Options"
GAS LISTING foo.s                       page 2
User's Guide Example
Listing Options
   4                 .psize 10
   5
   6 0000 01001A3C    lui $k0, 1
   7 0004 02001A3C    lui $k0, 2
   8 0008 03001A3C    lui $k0, 3
   9                 .eject
GAS LISTING foo.s                       page 3
User's Guide Example
Listing Options
  10 000c 04001A3C    lui $k0, 4
  11 0010 05001A3C    lui $k0, 5
```

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

```
   1              # 1 "foo.S"
   2              # 1 "<built-in>"
   1                 .text
   0
   0
   2                 .title "User's Guide Example"
   3                 .sbttl "Listing Options"
   4                 .psize 10
   5
   6 0000 01001A3C    lui $k0, 1
   7 0004 02001A3C    lui $k0, 2
   8 0008 03001A3C    lui $k0, 3
   9                 .eject
  10 000c 04001A3C    lui $k0, 4
  11 0010 05001A3C    lui $k0, 5
```

2.4.1.7 -as

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

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

```
GAS LISTING example.s page 1

DEFINED SYMBOLS
                           *ABS*:00000000 src\example.c
           example.s:18    .text:00000000 main
                           *COM*:00000004 testval

UNDEFINED SYMBOLS
bar
```

2.4.1.8 -a=*file*

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

2.4.2 --listing-lhs-width *num*

The --listing-lhs-width option is used to set the width of the output data column of the listing file. By default, this is set to 1 word. The following line is extracted from a listing. The output data column is in bold text.

```
2 0000 54686973  .ascii "This is an example"
2      20697320
2      616E2065
2      78616D70
2      6C650000
```

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

```
2 0000 54686973 20697320  .ascii "This is an example"
2      616E2065 78616D70
2      6C650000
```

2.4.3 --listing-lhs-width2 *num*

The --listing-lhs-width2 option is used to set the width of the continuation lines of the output data column of the listing file. By default, this is set to 1. If the specified width is smaller than the first line, this option is ignored. The following lines are extracted from a listing. The output data column is in bold.

```
2 0000 54686973  .ascii "This is an example"
2      20697320
2      616E2065
2      78616D70
2      6C650000
```

If the option --listing-lhs-width2 3 is used, then the same line will appear as follows in the listing:

```
2 0000 54686973  .ascii "This is an example"
2      20697320 616E2065 78616D70
2      6C650000
```

2.4.4 --listing-rhs-width *num*

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

```
2 0000 54686973  .ascii "This line is long"
2      206C696E
2      65206973
2      206C6F6E
2      67000000
```

If the option --listing-rhs-width 22 is used, then the same line will appear as follows in the listing:

```
2 0000 54686973  .ascii "This line is
2      206C696E
2      65206973
2      206C6F6E
2      67000000
```

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

2.4.5 --listing-cont-lines *num*

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

```
2 0000 54686973  .ascii "This is a long character
                         sequence"
2      20697320
2      61206C6F
2      6E672063
2      68617261
```

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

```
2 0000 54686973  .ascii "This is a long character
                         sequence"
2      20697320
2      61206C6F
```

2.5 OPTIONS THAT CONTROL INFORMATIONAL OUTPUT

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

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

2.5.1 `--fatal-warnings`

Warnings are treated as if they were errors.

2.5.2 `--no-warn (-W)`

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

2.5.3 `--warn`

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

2.5.4 `-J`

No warnings are issued about signed overflow.

2.5.5 `--help`

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

2.5.6 `--target-help`

The assembler will show a message regarding the PIC32 target-specific command-line options. The assembler then exits.

2.5.7 `--version`

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

2.5.8 `--verbose (-v)`

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

2.6 OPTIONS THAT CONTROL OUTPUT FILE CREATION

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

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

2.6.1 `-g`

Generate symbolic debugging information.

2.6.2 `--keep-locals (-L)`

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

2.6.3 `-o` *`objfile`*

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

2.6.4 `-Z`

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

2.6.5 `-MD` *`file`*

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

2.7 ASSEMBLER SYMBOL-DEFINITION AND SEARCH-PATH OPTIONS

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

2.7.1 --defsym *sym=value*

Define symbol *sym* to given *value*.

2.7.2 -I *dir*

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

When passed directly to the assembler, this option affects the search path used by the assembler's `.include` directive. To affect the search path used by the C preprocessor for a `#include` directive, pass the corresponding option to the xc32-gcc compilation driver.

2.8 COMPILATION-DRIVER AND PREPROCESSOR OPTIONS

The compilation-driver (xc32-gcc) and C preprocessor options in this section may be useful for assembly-code projects. The compilation driver will pass the options to the preprocessor as required. See the "*MPLAB XC32 C/C++ Compiler User's Guide*" (DS51686) for more information on the compilation driver and for a more comprehensive list of driver options.

2.8.1 `-mprocessor=`*`device`*

Selects the device for which to compile (e.g., `-mprocessor=32MX360F512L`).

2.8.2 `-Wa,`*`option`*

Pass *`option`* as an option to the assembler. If *`option`* contains commas, it is split into multiple assembler options at the commas. The *`option`* argument must not contain white space.

2.8.3 `-Dmacro=`*`defn`*

Define macro macro as *`defn`*. All instances of `-D` on the command line are processed before any `-U` options.

2.8.4 `-Dmacro`

Define macro macro as 1. All instances of `-D` on the command line are processed before any `-U` options.

2.8.5 `-Umacro`

Undefine macro macro. `-U` options are evaluated after all `-D` options, but before any `-include` and `-imacros` options.

2.8.6 `-I` *`dir`*

Add the directory *`dir`* to the head of the list of directories to be searched for `#include` preprocessor header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I` option, the directories are scanned in left-to-right order. The standard system directories come after.

When passed to the compilation driver, this option affects the search path used by the preprocessor's `#include` directive. To affect the search path used by the assembler's `.include` directive, pass the corresponding option to the assembler using the `-Wa` option.

2.8.7 `-save-temps`

Do not delete intermediate files. Place them in the current directory and name them based on the source file. Thus, compiling `foo.c` with `-c -save-temps` would produce the following files:

- `foo.i` (preprocessed file)
- `foo.s` (assembly language file)
- `foo.o` (object file)

2.8.8 `-v`

Print the commands executed during each stage of compilation.

2.8.9 `--help`

Print a description of the command-line options.

Chapter 3. MPLAB XC32 Assembly Language

3.1 INTRODUCTION

The source language accepted by the macro assembler is described here. All opcode mnemonics and operand syntax are specific to the target device. The same assembler application is used for compiler-generated intermediate assembly and hand-written assembly source code.

Topics covered in this chapter are:

- Internal Preprocessor
- Source Code Format
- Special Characters
- Symbols
- Giving Symbols Other Values
- The Special DOT Symbol
- Expressions
- Operators
- Special Operators

3.2 INTERNAL PREPROCESSOR

The assembler has an internal preprocessor. The internal processor performs the following actions.

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

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

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

The internal processor does **not** perform the following actions.

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

You can do include file preprocessing with the `.include` directive. See **Chapter 4. "Assembler Directives"**.

You can use the C compiler driver to get other C-style preprocessing by giving the input file a `.S` suffix. See the "*MPLAB XC32 C/C++ Compiler User's Guide"* (DS51686) for more information.

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

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

3.3 SOURCE CODE FORMAT

Assembly source code consists of statements and white spaces.

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

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

```
[label:][mnemonic[operands] ][; comment]
```

OR

```
[label:][directive[arguments] ][; comment]
```

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

3.3.1 Label

A label is one or more characters chosen from the set composed of all letters, digits, the underline character (_), and the period (.). Labels may not begin with a decimal digit, except for the special case of a local symbol. (See **Section 3.5.1 "Local Symbols"** for more information.) Case is significant. There is no length limit; all characters are significant.

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

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

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

3.3.2 Mnemonic

Mnemonics tell the assembler which machine instructions to assemble. For example, addition (`ADD`), jumps (`J`), or loads (`LUI`). Unlike labels that you create yourself, mnemonics are provided by the PIC32 MCU assembly language. Mnemonics are not case sensitive.

See the data sheet of your target PIC32 MCU for more details on the CPU instruction-set mnemonics available for the device.

The assembler also supports a number of synthesized/macro instructions intended to make writing assembly code easier. The `LI` (load immediate) instruction is an example of a synthetic macro instruction. The assembler generates two machine instructions to load a 32-bit constant value into a register from this single synthetic instruction.

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

```
[label:] [mnemonic [operands] ] [; comment]
```

OR

```
[label:] [directive [arguments] ] [; comment]
```

3.3.3 Assembler Syntax

The assembler synthesizes instructions for:

• A 32-bit Load Immediate

• A load from a memory location

• An extended branch conditional

• A two-operand form of some three-operand instructions

• An unaligned load/store instruction

Assembly directives, such as `.set noat`, `.set nomacro`, and `.set noreorder`, disable these normally helpful features for cases where you require full control over the generated code. See **Section 4.13 "Directives that Control Code Generation"**.

3.3.4 Directive

Assembler directives are commands that appear in the source code but are not translated directly into machine code. Directives are used to control the assembler, its input, output and data allocation. The first character of a directive is a dot (`.`). More details are provided in **Chapter 4. "Assembler Directives"** on the available directives.

3.3.5 Operands

Each machine instruction takes from 0 up to 4 operands. (See the appropriate data sheet of your target PIC32 MCU for a full list of machine instructions.) These operands give information to the instruction on the data that should be used and the storage location for the instruction. Operands must be separated from mnemonics by one or more spaces or tabs.

Separate multiple operands with commas. If commas do not separate your operands, the assembler displays a warning and takes its best guess on the separation of the operands. For most PIC32 MCU instructions, an operand consists of a core general-purpose register, label, literal, or basereg+offset.

3.3.5.1 GENERAL-PURPOSE REGISTER OPERANDS

The PIC32 MCU core contains thirty-two 32-bit general purpose registers used for integer operations and address calculation. Most of the PIC32 MCU instructions require one or more GPR operands, either for the source, the destination, or both.

Register operands are distinguished with a preceding dollar sign ('$'). The register number immediately follows the dollar sign. Example 3-1 shows assembly source code using register number operands.

However, if you use the compilation driver (xc32-gcc) to preprocess the source code with the CPP-style preprocessor before assembling, you can take advantage of macros that are provided in the `xc.h` header file that is provided with the C compiler. These macros map conventional register names to the corresponding register number. Example 3-2 shows assembly source code using conventional register names for operands. See the "*MPLAB XC32 C/C++ Compiler User's Guide"* (DS51686) for additional information on PIC32 MCU register conventions and the compiler's runtime environment.

EXAMPLE 3-1: ASSEMBLY SOURCE CODE WITH REGISTER NUMBER OPERANDS

```
.text
# Add Word
li   $2, 123
li   $3, 456
add  $4, $2, $3
```

EXAMPLE 3-2: ASSEMBLY SOURCE CODE WITH CONVENTIONAL REGISTER NAMES

```
#include <xc.h>
  .text
  /* Add Word */
  li   v0, 123      /* v0 is a return-value register */
  li   v1, 456      /* v1 is a return-value register */
  add  a0, v0, v1   /* a0 is an argument register */
```

3.3.5.2 LITERAL-VALUE OPERANDS

Literal values can be hexadecimal, octal, binary, or decimal format. Hexadecimal numbers are distinguished by a leading 0x. Octal numbers are distinguished by a leading 0. Binary numbers are distinguished by a leading 0B or 0b. Decimal numbers require no special leading or trailing character.

Example:

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

`-5` represents the literal value -5.

`symbol` represents the value of `symbol`.

3.3.5.3 BASEREG+OFFSET OPERANDS

Load and store operations select the memory location using a BaseReg+Offset operand. For an operand of this type, the effective address is formed by adding the 32-bit signed offset to the contents of a base register. A PIC32 MCU data sheet shows this type of operand as `Mem[R+offset]`.

EXAMPLE 3-3: USING ASSEMBLY SOURCE CODE WITH BASEREG+OFFSET OPERANDS

```
#include <xc.h>
        .data
        .align 4
MY_WORD_DATA:
        .word 0x10203040, 0x8090a0b0
        .text
        .global example
        /* Store Word */
example:
        la   v0, MY_WORD_DATA
        lui         v1,0x1111
        ori         v1,v1,0x4432
        lui         a0,0x5555
        ori         a0,a0,0x1123
        sw v1, 0(v0)     /* Mem[GPR[v0]+0] <- GPR[v1] */
        sw a0, 4(v0)     /* Mem[GPR[v0]+4] <- GPR[a0] */
        lw a1, 0(v0)     /* GPR[a1] <- Mem[GPR[v0]+0] */
        b .
```

The C compiler supports global-pointer relative (gp-rel) addressing. Loads and stores to data lying within 32 KB of either side of the address stored in the gp register (64 KB total) can be performed in a single instruction using the gp register as the base register.

The C compiler's `-Gnum` option controls the maximum size of global and static data items that can be addressed in one instruction instead of two. The compiler's default gnum value is 8 bytes, which is large enough to hold all simple scalar variables.

> **Note:** To utilize gp-relative addressing, the compiler and assembler must group all of the "small" variables and constants into one of the "small" sections. See the *MPLAB® XC32 C/C++ Compiler User's Guide* (DS51686) for more information on the global pointer and the `-G` option.

EXAMPLE 3-4: ASSEMBLY SOURCE CODE WITH GP-RELATIVE ADDRESSING

```
    .align 2
    .globl foo
    .set   nomips32
    .ent   foo
foo:
    .set   noreorder
    .set   nomacro

    lw     $3,%gp_rel(testval)($28)
    addiu  $2,$3,1
    sw     $2,%gp_rel(testval)($28)
    j      $31
    nop

    .set   macro
    .set   reorder
    .end   foo
```

There are a few potential pitfalls to using gp-relative addressing:

- You must take special care when writing assembler code to declare global (i.e., public or external) data items correctly:
  - Writable, initialized data of gnum bytes or less must be put explicitly into the `.sdata` section, for example:

```
            .sdata
 small:     .word  0x12345678
```

  - Global common data must be declared with the correct size, for example:

```
            .comm small, 4
            .comm big, 100
```

  - Small external variables must also be declared correctly, for example:

```
            .extern smallext, 4
```

- If your program has a very large number of small data items or constants, the C compiler's `-G8` option may still try to push more than 64 KB of data into the "small" region; the symptom will be obscure relocation errors ("relocation truncated") when linking. Fix it by disabling gp-relative addressing with the compiler's `-G0` option and/or reducing the space reserved in the small data sections (i.e. `.sbss` and `.sdata)` in your assembly code.

3.3.6 Arguments

Each directive takes 0 to 3 arguments. These arguments give additional information to the directive on how it should carry out the command. Arguments must be separated from directives by one or more spaces or tabs. Commas must separate multiple arguments. More details on the available directives are provided in **Chapter 4. "Assembler Directives"**.

3.3.7 Comments

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

3.3.7.1 SINGLE LINE COMMENT

This type of comment extends from the comment character to the end of the line. For a single line comment, use a number/hash sign (#).

> **Note:** This comment character differs from the character recognized by the MPASM assembler and the MPLAB Assembler for PIC24 MCUs and dsPIC DSCs.

3.3.7.2 MULTI-LINE COMMENT

This type of comment can span multiple lines. For a multi-line comment, use `/* ... */`. These comments cannot be nested.

Example:

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

3.4 SPECIAL CHARACTERS

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

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

3.4.1 Numeric Constants

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

3.4.1.1 INTEGERS

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

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

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

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

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

3.4.1.2 FLOATING-POINT NUMBERS

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

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

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

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

3.4.2 Character Constants

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

3.4.2.1 CHARACTERS

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

TABLE 3-1: SPECIAL CHARACTERS AND USAGES

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

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

3.4.2.2 STRINGS

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

3.4.2.3 GENERAL SYNTAX RULES

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

TABLE 3-2: SYNTAX RULES

Character	Character Description	Syntax Usage
.	period	begins a directive
#	number/hash	begin single-line comment
/*	slash, asterisk	begin multiple-line comment
*/	asterisk, slash	end multiple-line comment
:	colon	end a label definition
	none required	begin a literal value
'c'	character in single quotes	specifies single character value
"string"	character string in double quotes	specifies a character string

3.5 SYMBOLS

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

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

- Local Symbols
- Giving Symbols Other Values
- The Special DOT Symbol
- Predefined Symbols

3.5.1 Local Symbols

Local symbols are used when temporary scope for a label is needed. There are ten local symbol names, which can be reused throughout the program. They may be referred to using the names ‘0’, ‘1’, ..., ‘9’. To define a local symbol, write a label of the form ‘N’, ‘N’, ..., ‘N’ (where N represents any digit 0-9). To refer to the most recent previous definition of that symbol, write ‘Nb’, using the same digit as when you defined the label. To refer to the next definition of a local label, write ‘Nf’. The ‘b’ stands for “backwards” and the ‘f’ stands for “forwards”.

There is no restriction on how you can use these labels, and you can reuse them too. You can repeatedly define the same local label (using the same number 'N'), although you can refer to only the most recently defined local label of that number (for a backwards reference) or the next definition of a specific local label for a forward reference.

Also note that the first 10 local labels (‘0:’. . . ‘9:’) are implemented in a slightly more efficient manner than the others.

Here is an example:

EXAMPLE 3-5: SYMBOL USAGE

```
1: b 1f
2: b 1b
1: b 2f
2: b 1b
```

Which is the equivalent of:

```
label_1: b label_3
label_2: b label_1
label_3: b label_4
label_4: b label_3
```

Local symbol names are only a notational device. They are immediately transformed into more conventional symbol names before the assembler uses them. These conventional symbol names are stored in the symbol table and appear in error messages and optionally emitted to the object file.

3.6 GIVING SYMBOLS OTHER VALUES

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign '=', followed by an expression.

Example:

```
VAR = 4
```

3.7 THE SPECIAL DOT SYMBOL

The special symbol '.' refers to the current address being processed by the assembler. Thus, the expression 'melvin: .long .' defines melvin to contain its own address. Assigning a value to . is treated the same as an .org directive. Thus, the expression '.=.+4' is the same as saying '.space 4'.

When used in an executable section, '.' refers to a Program Counter address. On a PIC32 MCU, the Program Counter increments by 4 for each 32-bit instruction word. User code should take care to properly align instructions after modifying the dot symbol.

3.7.1 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign '=', followed by an expression.

Example:

```
VAR = 4
```

3.7.2 The Special DOT Symbol

The special symbol '.' refers to the current address being processed by the assembler. Thus, the expression 'melvin: .long .' defines melvin to contain its own address. Assigning a value to . is treated the same as an .org directive. Thus, the expression '.=.+4' is the same as saying '.space 4'.

When used in an executable section, '.' refers to a Program Counter address. On a PIC32 MCU, the Program Counter increments by 4 for each 32-bit instruction word. User code should take care to properly align instructions after modifying the dot symbol.

3.7.3 Predefined Symbols

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

TABLE 3-3: PREDEFINED SYMBOLS

Symbol	Definition
P32MX	PIC32MX target device family
P32MZ	PIC32MZ target device family
HAS_MIPS32R2	Device supports the MIPS32r2 Instruction Set
HAS_MIPS16	Device supports the MIPS16e Instruction Set
HAS_MICROMIPS	Device supports the microMIPS Instruction Set
HAS_DSPR2	Device supports the DSPr2 engine
HAS_MCU	Device supports the MIPS MCU extensions
HAS_L1CACHE	Device has an L1 data and program cache
HAS_VECTOROFFSETS	Device uses configurable offsets for the vector table

3.8 EXPRESSIONS

An expression specifies an address or numeric value. White space may precede and/or follow an expression. The result of an expression must be an absolute number or an offset into a particular section. When an expression is not absolute and does not provide enough information for the assembler to know its section, the assembler terminates and generates an error message.

3.8.1 Empty Expressions

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

3.8.2 Integer Expressions

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

Integer expressions involving symbols in program memory are evaluated in Program Counter (PC) units. In MIPS32 mode, the Program Counter increments by 4 for each instruction word. For example, to branch to the next instruction after label L, specify L+4 as the destination.

Example:

```
b L+4
```

3.9 OPERATORS

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

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

3.9.1 Prefix Operators

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

TABLE 3-4: PREFIX OPERATORS

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

3.9.2 Infix Operators

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

TABLE 3-5: INFIX OPERATORS

Operator	Description	Example
Arithmetic		
*	Multiplication	`5 * 4 (=20)`
/	Division. Truncation is the same as the C operator '/'.	`23 / 4 (=5)`
%	Remainder	`30 % 4 (=2)`
<<	Shift Left. Same as the C operator '<<'	`2 << 1 (=4)`
>>	Shift Right. Same as the C operator '>>'	`2 >> 1 (=1)`
Bit-Wise		
&	Bit-wise And	`4 & 6 (=4)`
^	Bit-wise Exclusive Or	`4 ^ 6 (=2)`
!	Bit-wise Or Not	`0x1010 ! 0x5050 (=0xBFBF)`
|	Bit-wise Inclusive Or	`2 | 4 (=6)`
Simple Arithmetic		
+	Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.	`4 + 10 (=14)`
-	Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.	`14 - 4 (=10)`
Relational		
==	Equal to	`.if (x == y)`
!=	Not equal to (also <>)	`.if (x != y)`
<	Less than	`.if (x < 5)`
<=	Less than or equal to	`.if (y <= 0)`
>	Greater than	`.if (x > a)`
>=	Greater than or equal to	`.if (x >= b)`
Logical		
&&	Logical AND	`.if ((x > 1) && (x < 10))`
||	Logical OR	`.if ((y != x) || (y < 100))`

3.10 SPECIAL OPERATORS

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

- Obtaining the Size of a Specific Section
- Obtaining the Starting Address of a Specific Section
- Obtaining the End Address of a Specific Section

DD

TABLE 3-6: SPECIAL OPERATORS

Operators	Description
`.sizeof.(`*name*`)`	Get size of section *name* in address units
`.startof.(`*name*`)`	Get starting address of section *name*
`.endof.(name)`	Get ending address of section *name*

3.10.1 Obtaining the Size of a Specific Section

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

```
.word .sizeof.(.data)
```

3.10.2 Obtaining the Starting Address of a Specific Section

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

```
.word .sizeof.(.data)
```

3.10.3 Obtaining the Ending Address of a Specific Section

The .endof.(section_name) operator can be used to obtain the ending address of a specific section after the link process has occurred. For example, to obtain the starting address of the .data section, use:

```
.word .sizeof.(.data)
```

NOTES:

Chapter 4. Assembler 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:** Assembler directives are ***not*** target instructions (`ADD`, `XOR`, `JAL`, etc). For instruction set information, consult your target-device data sheet

While there are many significant similarities with directives supported by the 16-bit MPLAB Assembler for PIC24 MCUs and dsPIC DSCs (xc16-as), there are many differences in the directive set supported by the 32-bit MPLAB XC32 Assembler (xc32-as).

Topics covered in this chapter are:

- Directives that Define Sections
- Directives that Initialize Constants
- Directives that Declare Symbols
- Directives that Define Symbols
- Directives that Modify Section Alignment
- Directives that Format the Output Listing
- Directives that Control Conditional Assembly
- Directives for Substitution/Expansion
- Directives that Include Other Files
- Directives that Control Diagnostic Output
- Directives for Debug Information
- Directives that Control Code Generation

4.2 DIRECTIVES THAT DEFINE SECTIONS

Sections are locatable blocks of code or data that will occupy contiguous locations in the 32-bit device memory. Three sections are pre-defined: `.text` for executable code, `.data` for initialized data and `.bss` for uninitialized data. Other sections may be defined; the linker defines several that are useful for locating data in specific areas of 32-bit memory.

Section directives are:

- `.bss`
- `.data`
- `.pushsection name`
- `.popsection`
- `.section name [, "flags"] (deprecated)`
- `.section name [, attr1[,…,attrn]]`
- `.text`

.bss

Definition

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

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

Use the `.bss` directive to switch into the bss section and then define symbols as usual. You may assemble only zero values into the section. Typically the section will contain only symbol definitions and `.skip` directives

Example

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

.data

Definition

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

Example

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

The linker collects initial values for section `.data` (and other sections defined with the data attribute) and creates a data initialization template. This template can be processed during application start-up to transfer initial values into memory. For C applications, a library function is called for this purpose automatically. Assembly projects can utilize this library by linking with the libpic32 library. For more information, see the discussion of **Section 9.5.3 "Run-Time Library Support"** in **Initialized Data**.

.pushsection *name*

This directive pushes the current section onto the top of the section stack and then replaces the current section with *name*. Every .pushsection should have a matching .popsection.

.popsection

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

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

.section name [, attr1[,...,attrn]]

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

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

If the optional argument is not present, the section attributes depend on the section name. A table of reserved section names with implied attributes is given in Reserved Section Names with Implied Attributes. If the section name matches a reserved name,

the implied attributes will be assigned to that section. If the section name is not recognized as a reserved name, the default attribute will be data (initialized storage in data memory).

Implied attributes for reserved section names other than [.text, .data, .bss] are deprecated.

A warning will be issued if implied attributes for these reserved sections are used.

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

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

4.2.1 Attributes That Represent Section Types

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

TABLE 4-1: ATTRIBUTES THAT REPRESENT SECTION TYPES

Attribute	Description
code	Executable code in program memory
data	Initialized storage in data memory
bss	Uninitialized storage in data memory
persist	Persistent storage in data memory
ramfunc	Function in data memory

4.2.2 Attributes that Modify Section Types

Depending on the attribute, all or some section types may be modified by it, as below Table Table 4-2 in word file

TABLE 4-2: ATTRIBUTES THAT MODIFY SECTION TYPES

Attribute	Description	Attribute applies to:				
		code	data	bss	persist	ramfunc
address(a)	Locate at absolute address a	x	x	x	x	
near	Locate in the first 64k of memory		x	x	x	
reverse	Align the ending address +1		x	x	x	
align(n)	Align the starting address	x	x	x	x	x
noload	Allocate, do not load	x	x	x	x	x
keep	Keep section against garbage collection	x	x	x	x	x

4.2.3 Combining Attributes that Modify Section Types

TABLE 4-3: COMBINING ATTRIBUTES THAT MODIFY SECTION TYPES

	address	near	reverse	align	noload	keep
address		x	x		x	x
near	x		x	x	x	x
reverse		x			x	x
align	x	x			x	x
noload	x	x	x	x		x
keep	x	x	x	x	x	

The following section names are available for user applications:

TABLE 4-4: RESERVED SECTION NAMES

Section Name	Generated by	Mapped in the linker script to	Implied Attributes
.text	Compiler- or assembler generated instructions		code
.text.*	Functions when compiled with -ffunction-sections are output to uniquely named sections of this form		code
.startup	C start-up code/ left in the linker script for backwards compatibility	kseg0_boot_mem	code
.app_excpt	General-Exception handler	kseg0_boot_mem	code
.reset	Reset handler	kseg0_boot_mem	code
.bev_excpt	BEV-Exception handler	kseg0_boot_mem	code
.vector_n	Interrupt Vector n	kseg0_boot_mem	code
.rodata	Strings and C data declared const		code
.rodata.*	Constant data when compiled with -fdata-sections are output to uniquely named sections of this form		code
.data	Variables >n bytes (compiled -Gn) with an initial value.		data
.data.*	Large initialized variables compiled with -fdata-sections		data

TABLE 4-4: RESERVED SECTION NAMES (CONTINUED)

.ramfunc	RAM-functions		data
.bss	Uninitialized data		bss
.lit4 / .lit8	Constants (usually floating point) which the assembler decides to store in memory rather than in the instruction stream. Used for gp-relative addressing.		data
.sdata	Variables <=n bytes (compiled -Gn) with an initial value. Used for gp-relative addressing.		data
.sdata.*	Small variables compiled with -fdata-sections. Used for gp-relative addressing		data
.sbss	Uninitialized variables <=n bytes (compiled -Gn). Used for gp-relative addressing.		data
.sbss.*	Small uninitialized variables compiled with -fdata-sections. Used for gp-relative addressing.		data
.bss	Uninitialized larger variables		data
.bss.*	Uninitialized variables compiled with -fdata-sections.		data
.heap	Heap used for dynamic memory		data
.stack	Minimum space reserved for stack		data
.debug*	DWARF debug information		info
.line	DWARF debug information		info
.comment	#ident/.ident strings		info
.reginfo	Information section		info

Section Directive Examples

```
.section foo                          ;foo is initialized data memory.
.section bar,bss,align(256)           ;bar is uninitialized data memory, aligned.
.section *,data,near                  ;section is near initialized data memory.
.section buf1,bss,address(0xa0000800);buf1 is uninitialized data memory at 0xa0000800.
.section *, code                      ;section is in program memory
```

.text

Definition

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

Example

```
            .text
            .ent _main_entry
_main_entry:
            jal main
            nop
            jal exit
            nop
1:
            b  1b
            nop
            .end _main_entry
```

4.3 DIRECTIVES THAT INITIALIZE CONSTANTS

Constant initialization directives are:

- .ascii "string$_1$" [, ..., "string$_n$"]
- .asciz "string$_1$" [, ..., "string$_n$"]
- .byte expr$_1$[, ..., expr$_n$]
- .double value$_1$[, ..., value$_n$]
- .float value$_1$[, ..., value$_n$]
- .single value$_1$[, ..., value$_n$]
- .hword expr$_1$[, ..., expr$_n$]
- .int expr$_1$[, ..., expr$_n$]
- .long expr$_1$[, ..., expr$_n$]
- .short expr$_1$[, ..., expr$_n$]
- .string "str"
- .word expr$_1$[, ..., expr$_n$]

.ascii "*string*$_1$" [, ..., "*string*$_n$"]

.ascii expects zero or more string literals separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

.asciz "*string*$_1$" [, ..., "*string*$_n$"]

.asciz is just like .ascii, but each string is followed by a zero byte. The "z" in .asciz stands for "zero". This directive is a synonym for .string.

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

.byte expects zero or more expressions, separated by commas. Each expression is assembled into the next byte in the current section.

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

Assembles one or more double-precision (64-bit) floating-point constants into consecutive addresses in little-endian format. Floating point numbers are in IEEE format (see **Section 3.4.1.2 "Floating-Point Numbers"**).

The following statements are equivalent:

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

Alternatively, you can specify the hexadecimal encoding of a floating-point constant. The following statements are equivalent and encode the value 12345.67 as a 64-bit double-precision number:

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

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

Assembles one or more single-precision (32-bit) floating-point constants into consecutive addresses in little-endian format. It has the same effect as `.single`. Floating point numbers are in IEEE format (see **Section 3.4.1.2 "Floating-Point Numbers"**).

The following statements are equivalent:

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

Alternatively, you can specify the hexadecimal encoding of a floating-point constant. The following statements are equivalent and encode the value 12345.67 as a 32-bit double-precision number:

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

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

Assembles one or more single-precision (32-bit) floating-point constants into consecutive addresses in little-endian format. This directive is a synonym for `.float`. Floating point numbers are in IEEE format (see **Section 3.4.1.2 "Floating-Point Numbers"**).

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

Assembles one or more 2-byte numbers into consecutive addresses in little-endian format. This directive is a synonym for `.short`.

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

Assembles one or more 4-byte numbers into consecutive addresses in little-endian format. This directive is a synonym for `.long`.

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

Assembles one or more 4-byte numbers into consecutive addresses in little-endian format. This directive is a synonym for `.int`.

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

Assembles one or more 2-byte numbers into consecutive addresses in little-endian format. This directive is a synonym for `.hword`.

.string "*str*"

This directive is a synonym for `.asciz`.

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

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

4.4 DIRECTIVES THAT DECLARE SYMBOLS

Declare symbol directives are:

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

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

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

The `.comm` directive takes an optional third argument. If *algn* is specified, it is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be zero). The alignment must be an absolute expression, and it must be a power of two. If linker allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, the assembler will set the alignment to the largest power of two less than or equal to the size of the symbol, up to a maximum of 1.

.extern *symbol*

The `.extern` directive declares a symbol name that may be used in the current module, but it is defined as global in a different module. However, all symbols are `extern` by default so this directive is optional.

.global *symbol*
.globl *symbol*

The `.global` directive declares a symbol that is defined in the current module and is available to other modules. `.global` makes the symbol visible to the linker. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program.

Both spellings (`.globl` and `.global`) are accepted, for compatibility with other assemblers.

.lcomm *symbol*, *length*

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

.weak *symbol*

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

4.5 DIRECTIVES THAT DEFINE SYMBOLS

Define symbol directives are:

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

.equ *symbol, expression*

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

.equiv *symbol, expression*

Like `.equ`, except that the assembler will signal an error if symbol is already defined. Note that a symbol which has been referenced but not actually defined is considered to be undefined.

Except for the contents of the error message, this directive is roughly equivalent to:

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

4.6 DIRECTIVES THAT MODIFY SECTION ALIGNMENT

Directives that explicitly modify section alignment are listed below.

> **Note:** User code must take care to properly align an instruction following a directive that modifies the section alignment or location counter.

- `.align [algn[, fill]]`
- `.fill repeat[, size[, value]]`
- `.org new-lc[, fill]`
- `.skip size[, fill]`
- `.space size[, fill]`
- `.struct expression`

.align [*algn*[, *fill*]]

The `.align` directive pads the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required specified as the number of low-order zero bits the location counter must have after advancement.

The assembler accepts *`algn`* values from 0 up to 15. A `.align` 0 turns off the automatic alignment used by the data creating pseudo-ops. You must make sure that data is properly aligned. Reinstate auto alignment with a `.align` pseudo instruction.

The second expression (also absolute) gives the *`fill`* value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero by default. You may wish to use 0xFF for FLASH regions of memory.

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

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

`size` is optional. If the first comma and following tokens are absent, `size` is assumed to be 1.

`value` is optional. If the second comma and *`value`* are absent, `value` is assumed zero.

Example:

```
          .text
          .fill 0x3, 1, 0xFF
          .align 2
mylabel:  b .
```

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

The .org directive advances the location counter of the current section to *new-lc*. *new-lc* is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use .org to cross sections: if *new-lc* has the wrong section, the .org directive is ignored. If the section of *new-lc* is absolute, xc32-as issues a warning, then pretends the section of *new-lc* is the same as the current subsection.

.org may only increase the location counter, or leave it unchanged; you cannot use .org to move the location counter backwards.

Because the assembler tries to assemble programs in one pass, *new-lc* may not be undefined.

Beware that the origin is relative to the start of the section, not to the start of the subsection.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with *fill*, which should be an absolute expression. If the comma and *fill* are omitted, fill defaults to zero.

.skip *size*[, *fill*]

.space *size*[, *fill*]

These directives emit *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

.struct *expression*

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

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

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

4.7 DIRECTIVES THAT FORMAT THE OUTPUT LISTING

Output listing format directives are:

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

.eject

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

.list

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

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

.nolist

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

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

.psize *lines*[, *columns*]

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

If you do not use `.psize`, listings use a default line count of 60. You may omit the comma and columns specification; the default width is 200 columns.

The assembler generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify lines as 0, no formfeeds are generated save those explicitly specified with `.eject`.

.sbttl "*subheading*"

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

.title "*heading*"

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

4.8 DIRECTIVES THAT CONTROL CONDITIONAL ASSEMBLY

Conditional assembly directives are:

- `.else`
- `.elseif expr`
- `.endif`
- `.if expr`

.else

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

.elseif *expr*

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

.endif

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

.if *expr*

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

The assembler also supports the following variants of `.if`.

.ifdecl *symbol*

Assembles the following section of code if the specified symbol has been defined. Note that a symbol which has been referenced, but not yet defined, is considered to be undefined.

.ifc *string1*,*string2*

This directive assembles the following section of code if the two strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings which contain whitespace should be quoted. The string comparison is case sensitive.

.ifeq *absolute-expression*

This directive assembles the following section of code if the argument is zero.

.ifeqs *string1*,*string2*

This directive is another form of `.ifc`. The strings must be quoted using double quotes.

.ifge *absolute-expression*

This directive assembles the following section of code if the argument is greater than or equal to zero.

.ifgt *absolute-expression*

This directive assembles the following section of code if the argument is greater than zero.

.ifle *absolute-expression*

This directive assembles the following section of code if the argument is less than or equal to zero.

.iflt *absolute-expression*

This directive assembles the following section of code if the argument is less than zero.

.ifnc *string1*,*string2*

This directive is like `.ifc`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

.ifndef *symbol*

This directive assembles the following section of code if the specified symbol has not been defined. Both spelling variants are equivalent. Note a symbol which has been referenced but not yet defined is considered to be undefined.

.ifnotdef *symbol*

This directive is the same as `.ifndef`.

.ifne *absolute-expression*

This directive assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to `.if`).

.ifnes *string1*,*string2*

This directive is like `.ifeqs`, but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

4.9 DIRECTIVES FOR SUBSTITUTION/EXPANSION

Substitution/expansion directives are:

- `.exitm`
- `.irp symbol, value1 [, ..., valuen] ... .endr`
- `.irpc symbol, value ... .endr`
- `.macro`

.exitm

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

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

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

For example, assembling

```
.irp reg,0,1,2,3
lw $a\reg, 1032+\reg($sp)
.endr
```

is equivalent to assembling

```
lw $a0,1032+0($sp)
lw $a1,1032+1($sp)
lw $a2,1032+2($sp)
lw $a3,1032+3($sp)
```

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

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

For example, assembling

```
.irpc reg,0123
lw $a\reg, 1032+\reg($sp)
.endr
```

is equivalent to assembling

```
lw $a0,1032+0($sp)
lw $a1,1032+1($sp)
lw $a2,1032+2($sp)
lw $a3,1032+3($sp)
```

.macro

The directives .macro and .endm allow you to define macros that generate assembly output. For example, this definition specifies a macro SUM that puts a sequence of numbers into memory:

```
.macro SUM from=0, to=5
.long \from
.if    \+o-\from
SUM "(\from+1)",   \+o
.endif
.endm
```

With that definition, 'SUM 0,5' is equivalent to this assembly input:

```
.long 0
.long 1
.long 2
.long 3
.long 4
.long 5
```

.macro *macname*

.macro *macname macargs …*

Begin the definition of a macro called *macname*. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with =deflt. For example, these are all valid .macro statements:

- .macro comm
  Begin the definition of a macro called comm, which takes no arguments.
- .macro plus1 p, p1
  .macro plus1 p p1
  Either statement begins the definition of a macro called plus1, which takes two arguments; within the macro definition, write \p or \p1 to evaluate the arguments.
- .macro reserve_str p1=0 p2
  Begin the definition of a macro called reserve_str, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as 'reserve_str a,b' (with \p1 evaluating to a and \p2 evaluating to b), or as 'reserve_str ,b' (with \p1 evaluating as the default, in this case '0', and \p2 evaluating to b).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, 'SUM 9,17' is equivalent to 'sum to=9, from=17'.

.endm

Mark the end of a macro definition.

.exitm

Exit early from the current macro definition.

\@

The assembler maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with `\@`, but only within a macro definition. In the following example, a recursive macro is used to allocate an arbitrary number of labeled buffers

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

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

This example macro expands as shown in the following listing:

```
   6            make_buffers (\num - 1),\size
   7            .endif
   8            .endm
   9
  10            .bss
  11            # create BUF0..BUF3, 16 bytes each
  12            make_buffers 4,16
  12     > BUF0:.space 16
  12 0000 > .space 16
  12     > .if (4-1)
  12     > make_buffers (4-1),16
  12     >> BUF1:.space 16
  12 0010 >> .space 16
  12     >> .if ((4-1)-1)
  12     >> make_buffers ((4-1)-1),16
  12     >>> BUF2:.space 16
  12 0020 >>> .space 16
  12     >>> .if (((4-1)-1)-1)
  12     >>> make_buffers (((4-1)-1)-1),16
  12     >>>> BUF3:.space 16
  12 0030 >>>> .space 16
  12     >>>> .if ((((4-1)-1)-1)-1)
  12     >>>> make_buffers ((((4-1)-1)-1)-1),16
  12     >>>> .endif
  12     >>> .endif
  12     >> .endif
  12     > .endif
```

.purgem "*name*"

Undefine the macro name, so that later uses of the string will not be expanded. See `.macro` directive on the preceding page.

.rept *count* ... .endr

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

For example, assembling

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

is equivalent to assembling

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

4.10 DIRECTIVES THAT INCLUDE OTHER FILES

Directives that include data from other files are:

- `.incbin "file"[,skip[,count]]`
- `.include "file"`

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

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

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

.include "*file*"

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

4.11 DIRECTIVES THAT CONTROL DIAGNOSTIC OUTPUT

Miscellaneous directives are:

- `.abort`
- `.err`
- `.error "string"`
- `.fail expression`
- `.ident "comment"`
- `.print "string"`
- `.version "string"`
- `.warning "string"`

.abort

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

.err

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

.error "*string*"

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

.fail *expression*

Generates an error or a warning. If the value of the *`expression`* is 500 or more, `xc32-as` will print a warning message. If the value is less than 500, `as` will print an error message. The message will include the value of *`expression`*. This can occasionally be useful inside complex nested macros or conditional assembly.

.ident "*comment*"

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

.print "*string*"

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

.version "*string*"

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

.warning "*string*"

Similar to the directive `.error`, but emits a warning.

4.12 DIRECTIVES FOR DEBUG INFORMATION

Debug information directives are:

- `.ent function`
- `.end`
- `.file fileno "filename"`
- `.fmask mask, offset`
- `.frame framereg, frameoffset, retreg`
- `.loc fileno, lineno [columnno]`
- `.mask mask, offset`
- `.size name, expression`
- `.sleb128 expr`$_1$ `[, ..., expr`$_n$`]`
- `.type name, description`
- `.uleb128 expr1[,...,exprn]`

.ent *function*

This directive marks the `function` symbol as a function similarly to the generic `.type` directive.

.end

End program.

.file *fileno* "*filename*"

When emitting dwarf2 line-number information `.file` assigns filenames to the `.debug_line` file name table. The *`fileno`* operand should be a unique positive integer to use as the index of the entry in the table. The *`filename`* operand is a C string literal.

The detail of *`filename`* indices is exposed to the user because the filename table is shared with the `.debug_info` section of the dwarf2 debugging information, and thus the user must know the exact indices that table entries will have.

.fmask *mask*, *offset*

Not used for current PIC32 MCUs. Maintain `mask 0x00000000` and `offset 0`.

.frame *framereg*, *frameoffset*, *retreg*

This directive describes the shape of the stack frame. The virtual frame pointer in use is *`framereg`*; normally this is either $fp or $sp. The frame pointer is *`frameoffset`* bytes below the canonical frame address (CFA), which is the value of the stack pointer on entry to the function. The return address is initially located in *`retreg`* until it is saved as indicated in `.mask`.

.loc *fileno*, *lineno* [*columnno*]

The object file's debugging information contains a line-number matrix that correlates an assembly instruction to a line and column of source code. The `.loc` directive will add a matrix row corresponding to the assembly instruction immediately following the directive. The *`fileno`*, *`lineno`*, and *`columnno`* will be applied to the debug state machine before the row is added.

.mask *mask, offset*

Indicate which of the integer registers are saved in the current function's stack frame. *mask* is interpreted a bit mask in which bit n set indicates that register n is saved. The registers are saved in a block located *offset* bytes from the canonical frame address (CFA), which is the value of the stack pointer on entry to the function.

.size *name, expression*

This directive sets the size associated with a symbol *name*. The size in bytes is computed from *expression* which can make use of label arithmetic. This directive is typically used to set the size of function symbols.

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

sleb128 stands for "signed little endian base 128." This is a compact, variable-length representation of numbers used by the DWARF symbolic-debugging format.

.type *name, description*

This sets the type of symbol *name* to be either a function symbol or an object symbol. There are five different syntaxes supported for the type *description* field, in order to provide compatibility with various other assemblers. The syntaxes supported are:

```
.type <name>,#function
.type <name>,#object
.type <name>,@function
.type <name>,@object
.type <name>,%function
.type <name>,%object
.type <name>,"function"
.type <name>,"object"
.type <name> STT_FUNCTION
.type <name> STT_OBJECT
```

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

uleb128 stands for "unsigned little endian base 128." This is a compact, variable-length representation of numbers used by the DWARF symbolic-debugging format.

4.13 DIRECTIVES THAT CONTROL CODE GENERATION

Directives controlling assembler code-generation behavior are:

- `.set noat`
- `.set at`
- `.set noautoextend`
- `.set autoextend`
- `.set nomacro`
- `.set macro`
- `.set mips16e`
- `.set nomips16e`
- `.set noreorder`
- `.set reorder`

.set noat

When synthesizing some address formats, the assembler may require a scratch register. By default, the assembler will quietly use the at ($1) register, which is reserved as an assembler temporary by convention. In some cases, the compiler should not use that register. The `.set noat` directive prevents the assembler from quietly using that register.

.set at

Allow the assembler to quietly use the at ($1) register.

.set noautoextend

By default, MIPS16 instructions are automatically extended to 32 bits when necessary. The directive `.set noautoextend` will turn this off. When `.set noautoextend` is in effect, any 32-bit instruction must be explicitly extended with the `.e` modifier (e.g., `li.e $4,1000`). The directive `.set autoextend` may be used to once again automatically extend instructions when necessary.

.set autoextend

Enable auto-extension of MIPS16 instructions to 32 bits.

.set nomacro

The assembler supports synthesized instructions, an instruction mnemonic that synthesizes into multiple machine instructions. For instance, the `sleu` instruction assembles into an `sltu` instruction and an `xori` instruction. The `.set nomacro` directive causes the assembler to emit a warning message when an instruction expands into more than one machine instruction.

.set macro

Suppress warnings for synthesized instructions.

.set mips16e

Assemble with the MIPS16e ISA extension.

.set nomips16e

Do not assemble with the MIPS16e ISA extension.

.set noreorder

By default, the assembler attempts to fill a branch or delay slot automatically by reordering the instructions around it. This feature can be very useful.

Occasionally, you'll want to retain precise control over your instruction ordering. Use the `.set noreorder` directive to tell the assembler to suppress this feature until it encounters a `.set reorder` directive.

.set reorder

Allow the assembler to reorder instructions to fill a branch or delay slot.

.set micromips

Assemble with the microMIPS ISA mode.

.set nomicromips

Do not assemble with the microMIPS ISA mode.

Chapter 5. Assembler Errors/Warnings/Messages

5.1 INTRODUCTION

MPLAB Assembler for PIC32 MCUs (xc32-as) generates errors, warnings and messages. A descriptive list of the most common diagnostic messages from the assembler is shown here.

Topics covered in this chapter are:

- Fatal Errors
- Errors
- Warnings
- Messages

5.2 FATAL ERRORS

The following errors indicate that an internal error has occurred in the assembler. Please contact Microchip Technology (http://support.microchip.com) for support if the assembler generates any of the fatal errors listed below. Be sure to provide full details about the source code and command-line options causing the error.

- Bad char = '%c'
- Bad defsym; format is --defsym name=value
- Bad return from bfd_install_relocation: %x
- Broken assembler. No assembly attempted.
- Can't allocate elf private section data: %s
- Can't continue
- Can't create group: %s
- Can't extend frag %u chars
- Can't open a bfd on stdout %s
- Can't start writing .mdebug section: %s
- Cannot write to output file
- Could not write .mdebug section: %s
- Dwarf2 is not supported for this object file format
- Emulations not handled in this configuration
- Error constructing %s pseudo-op table: %s
- Expr.c(operand): bad atof_generic return val %d
- Failed sanity check
- Failed to read instruction table %s\n
- Failed to set up debugging information: %s
- Index into stored_fixups[] out of bounds
- Inserting into symbol table failed: %s
- Internal: bad mips opcode (bits 0x%lx undefined): %s %s.
- Internal: bad mips opcode (mask error): %s %s.
- Internal: bad mips opcode (unknown extension operand type `+%c'): %s %s.
- Internal: bad mips opcode (unknown operand type `%c'): %s %s.
- Internal error, line %d, %s
- Internal error: unknown dwarf2 format
- Internal: can't hash `%s': %s
- Invalid abi -mabi=%s
- Invalid listing option `%c'
- Macros nested too deeply
- Missing emulation mode name
- Multiple emulation names specified
- No compiled in support for 64 bit object file
- No object file generated
- Rva not supported
- Rva without symbol
- Too many fixups
- Unrecognized emulation name `%s'

5.3 ERRORS

The errors listed below usually indicate an error in the assembly source code or command-line options passed to the assembler.

Symbol

.abort detected. Abandoning ship.

User error invoked with the `.abort` directive.

.else without matching .if

A `.else` directive was seen without a preceding .if directive.

.elseif after .else

A `.elseif` directive specified after a `.else` directive. Modify your code so that the `.elseif` directive comes before the `.else` directive.

.elseif without matching .if

A `.elseif` directive was seen without a preceding `.if` directive.

.endfunc missing for previous .func

A `.endfunc` directive is missing for a previous `.func` directive.

.endif without .if

A `.endif` directive was seen without a preceding `.if` directive.

.err encountered.

User error invoked with the `.err` directive.

.Ifeqs syntax error

Two comma-separated, double-quoted strings were not passed as arguments to the `.ifeqs` directive.

.Set pop with no .set push

Attempting to pop options off of an empty option stack. Use `.set push` before `.set pop`.

.Size expression too complicated to fix up

The `.size` expression can be constant or use label subtraction.

A

A bignum with underscores may not have more than 8 hex digits in any word.

A bignum constant must not have more than 8 hex digits in a single word.

A bignum with underscores must have exactly 4 words.

A bignum constant using the underscore notation must have exactly four 8-hexdigit parts.

Absolute sections are not supported.

This assembler does not support the absolute section command.

Alignment not a power of 2.

The alignment value must be a power of 2. Modify the alignment to be a power of 2.

Alignment too large: 15. Assumed.

An alignment greater than 15 was requested. The assembler automatically continues with a alignment value of 15.

Arg/static registers overlap.

A MIPS32 mode save/restore uses overlapping registers for args and statics.

Argument must be a string.

The argument to a .error or .warning directive must be a double-quoted string.

Attempt to allocate data in common section.

This directive attempts to allocate data to a section that isn't allocatable. Allocate the data to another section instead.

Attempt to get value of unresolved symbol *name*

The assembler could not get the value of an unresolved symbol.

Attempt to set value of section symbol.

Assignments to section symbols are not legal.

B

Backward ref to unknown label *label*:

The referenced label is either not seen or not defined here.

Bad .common segment name

Could not determine an appropriate alignment value for a `.comm` symbol. A previously seen `.comm` symbol may be incorrect.

Bad escaped character in string.

The string uses a non-standard backslash-escaped character.

Bad expression.

The expression type cannot be determined or an operand is not of the correct type for the expression type.

Bad floating literal: %s.

The token could not be converted to a floating-point value.

Bad floating-point constant: exponent overflow.

The token could not be converted to a floating-point value because of exponent overflow.

Bad floating-point constant: unknown error code=%d.

The token could not be converted to a floating-point value.

Bad format for ifc or ifnc.

The arguments to the ifc or ifnc directive are incorrect. They must be 2 comma-separated, double-quoted strings.

Bad or irreducible absolute expression.

The absolute expression had an unexpected operator type.

Bad register expression.

The DWARF debugging directive has an invalid register expression.

Bignum invalid.

The bignum value specified in the expression is not valid.

C

Can't parse register list.

In MIPS32 mode, the register list is invalid.

Can't resolve value for symbol `%s'.

The assembler could not get a real value for the symbol.

Constant too large.

When sign extending a constant offset from a base register, the constant was too large.

Could not skip to num in file filename

The skip parameter to the `.incbin` directive was invalid for the given file.

D

Duplicate else.

Each .if directive can have only up to one corresponding .else directive.

E

End of file inside conditional.

The assembler identified a missing conditional-end directive. Terminate the conditional before the end of the file.

End of macro inside conditional.

The assembler identified a missing macro-end directive. Terminate the macro before the end of the file.

Expected address expression.

The expression was illegal, absent, or bignum but it should have been a constant address.

Expected comma after %s.

The arguments for this directive must be separated by a comma.

Expected comma after name `%s' in .size directive.

The arguments for this directive must be separated by a comma.

Expected quoted string.

The argument should be a quoted string.

Expected simple number.

This argument must be a simple number.

Expected symbol name.

This argument must be a symbol name.

Expression out of range.

The expression is out of range for the directive or instruction (e.g. 32-bit value when a 32-bit value is expected)

Expression too complex.

The expression should be a symbol or constant.

F

File not found: %s.

The file specified to a directive (such as `.incbin`) could not be opened as specified.

File number %ld already allocated.

The file number passed to a .file directive is already in use.

File number less than one.

The file number passed to a `.file` directive must be > 1.

Floating point number invalid.

The floating-point number is invalid.

G

Global symbols not supported in common sections.

External symbols are not supported in MRI common sections.

I

Ignoring attempt to redefine symbol name

The symbol being redefined by the `.weakext` directive has already been defined.

Improper insert size

The width of the field specified to an INS instruction was not valid for the shift position.

Improper extract size

The width of the field specified to an EXT instruction was not valid for the shift position.

Instruction insn requires absolute expression.

This instruction requires a constant expression.

Invalid astatic register list

The aregs field of a MIPS32e extended SAVE/RESTORE instruction specified an invalid astatic register list.

Invalid arg register list.

The aregs field of a MIPS32e extended SAVE/RESTORE instruction specified an invalid arg register list.

Invalid coprocessor 0 register number.

An invalid coprocessor 0 register number was passed to this instruction.

Invalid coprocessor sub-selection value (%ld), not in range 0-7.

The coprocessor sub-selection value must be in the range 0-7.

Invalid frame size

The frame size is not valid and could not be encoded.

Invalid identifier for .ifdef.

The specified identifier is not a valid name. It must begin with a legal character.

Invalid register list.

In MIPS32 mode, the register list contained an invalid register.

Invalid segment %s.

Attempting to change the location counter in an invalid segment.

Invalid static register list.

The static register list should include only $s2-$s8

J

Jump to misaligned address (0x%lx).

The jump target address is not aligned.

Junk at end of line, first unrecognized character is char

There are extraneous characters after the expected input.

Junk at end of line, first unrecognized character valued 0xval

There are extraneous characters after the expected input.

L

Load/store address overflow (max 32 bits).

The load/store address is greater than 32 bits wide. Make sure that the label is correct.

Local label is not defined.

A referenced local label was never defined.

Lui expression not in range 0..65535.

The Load Upper Immediate expression should be within the 32-bit range.

N

New line in title.

The title heading string should be enclosed in double quotes.

No such section.

The section name specified in a `.global` directive does not exist. (e.g. `.global foo .myscn`)

Non-constant expression in .elseif statement

The `.elseif` statement requires a constant `expr` expression. The argument of the `.elseif` directive must be a constant value able to be resolved on the first pass of the directive. Ensure that any `.equ` of a symbol used in this argument is located before the directive. See **Section 4.8 "Directives that Control Conditional Assembly"** for more details.

Non-constant expression in .if statement.

The `.if` statement requires a constant `expr` expression. The argument of the `.if` directive must be a constant value able to be resolved on the first pass of the directive. Ensure that any `.equ` of a symbol used in this argument is located before the directive. See **Section 4.8 "Directives that Control Conditional Assembly"** for more details.

`Noreorder' must be set before `nomacro'.

Set noreorder before setting nomacro.

Number (0x%lx) larger than 32 bits.

Loading a value greater than 32 bits wide into a register.

Number larger than 64 bits.

Loading a value greater than 64 bits wide into HI/LO registers.

O

Offset too large.

The offset must be within the signed-extended 32-bit range.

Opcode not supported on this processor.

The instruction opcode is not supported on PIC32 MCUs.

Operand overflow.

The operand is not within the allowed range for this instruction.

Operation combines symbols in different segments.

The left-hand side of the expression and the right-hand side of the expression are located in two different sections. The assembler does not know how to handle this expression.

R

Register value used as expression.

An expression's operator is a register rather than a valid operator.

Relocation reloc isn't supported by the current ABI.

This relocation isn't supported by the PIC32 little-endian ELF output format.

S

Seek to end of .incbin file failed `%s'.

Could not find the end of the file specified by `.incbin`

Skip (%ld) + count (%ld) larger than file size (%ld).

The `.incbin` skip value + count value is greater than the size of the file.

Store insn found in delay slot of noreorder code.

Consider moving the store in front of the branch and using a nop in the delay instead.

Symbol `%s' can not be both weak and common.

Both the `.weak` directive and `.comm` directive were used on the same symbol within the same source file.

Symbol name is already defined.

The symbol cannot be redefined.

Symbol definition loop encountered at `%s'.

The symbol could not be defined because a self-referential loop was encountered. A symbol's definition cannot depend on its own value.

Syntax error in .startof. Or .sizeof.

The assembler found either `.startof.` or `.sizeof.`, but did not find the beginning parenthesis '(' or ending parenthesis ')'.

T

This string may not contain '\0'.

The string must be a valid C string and cannot contain '`\0`'.

Treating warnings as errors.

The assembler has been instruction to treat all warnings as errors with the `--fatal-warnings` command-line option.

U

Unassigned file number num

The `.loc` directive specifies a file number that is not yet in use.

Unclosed '('.

An open '(' is unmatched with a closing ')'. Add the missing ')'.

Unexpected register in list.

In MIPS32 mode, an invalid register was used. Check the operands for this instruction.

5.4 WARNINGS

The assembler generates warnings when an assumption is made so that the assembler could continue assembling a flawed program. Warnings should not be ignored. Each warning should be specifically looked at and corrected to ensure that the assembler understands what was intended. Warning messages can sometimes point out bugs in your program.

Symbol

.end directive missing or unknown symbol

The `.end` function debugging-info directive is missing or the associated symbol is not defined. Make sure that the `.end` directive is placed appropriately after the `.ent` directive.

.end directive without a preceding .ent directive.

The `.end` function debugging-info directive does not have an associated .ent directive to mark the symbol as a function. Make sure that the `.end` directive is positioned appropriately after a `.ent` directive.

.end not in text section

The `.end` function debugging-info directive must be in a section with executable code.

.end symbol does not match .ent symbol.

The `.end` function debugging-info directive's function argument does not match the preceding .ent directive's function argument. Make sure that the .end directive is positioned appropriately after its corresponding .ent directive.

.endr encountered without preceding .rept, .irc, or .irp

The `.endr` directive ends a `.rept`, `.irc`, or `.irp` sequence; however this `.endr` directive does not have a preceding `.rept`, `.irc`, or `.irp` directive. Make sure that the `.endr` directive is positioned correctly in your code.

.ent or .aent not in text section.

The `.ent` function debugging-info directive must be in a section containing executable code.

.fail *expr* encountered

If the value of the your `.fail` expression is 500 or more, the assembler will print a warning message. The message will include the value of expression.

.fill size clamped to 8

The `.fill` size value may be zero or more, but if it is more than 8, then it is deemed to have the value 8.

.frame outside of .ent

The `.frame` directive describes the stack frame and therefore must be used within a function.

.incbin count zero, ignoring filename

The `.incbin` count should be greater than zero. reading zero bytes from a file has no effect.

.mask/.fmask outside of .ent

The `.mask`/`.fmask` stack-frame information should be defined within a `.ent` function. Make sure that the `.mask`/`.fmask` directive is positioned correctly within the source code.

.popsection without corresponding .pushsection; ignored

The assembler cannot pop a section off of the section stack without pushing one onto the stack first.

.previous without corresponding .section; ignored

There's no previous section swap with the current section. Make sure that the `.previous` directive is positioned correctly within the source code.

.space repeat count is negative, ignored.

The `.space` size argument must be greater than 0.

.space repeat count is zero, ignored.

The `.space` size argument must be greater than 0.

A

Alignment negative: 0 assumed.

The `.align` alignment must be a non-negative power-of-two value. .align 0 turns off the automatic alignment used by the data creating pseudo-ops.

Alignment too large: 15 assumed.

The `.align` alignment value is greater than 15. The valid range is [0,15].

D

Divide by zero.

DIV instruction with $zero register as RT.

Division by zero.

This expression attempts to divide by zero. Check the operands.

E

Extended instruction in delay slot.

A MIPS32e extended instruction may not be placed in a jump delay slot as it will cause undefined behavior. Move the instruction out of the delay slot.

F

Floating point constant too large.

The hexadecimal encoding of a floating-point constant is too large. Make sure that your floating-point value is encoded correctly in the 32-bit or 64-bit IEEE format.

I

Ignoring changed section attributes for *name*

If section attributes are specified the second time the assembler sees a particular section, then they should be the same as the first time the assembler saw the section attributes. The assembler assumes that the first set of section attributes was correct.

Ignoring changed section entity size for *name*

The section entity size should be the same the second time the assembler sees a particular section. The assembler assumes that the section entity size the first time it saw the section was correct.

Ignoring changed section type for *name*

The section type should be the same the second time the assembler sees a particular section. The assembler assumes that the section type the first time it saw the section was correct.

Ignoring incorrect section type for *name*

When switching to a special predefined section by name, the section's type should match the predefined type. The assembler uses the predefined type for the section.

Immediate for %s not in range 0..1023 (%lu).

The debugger Break code was not in the valid range. Normal user code should not use this instruction reserved for debugger use.

Improper shift amount (%lu).

The shift value for a shift instruction (e.g. `SLL`, `SRA`, `SRL`) is out of range.

Instruction sne: Instruction %s: result is always false.

The result of the condition tested by the `SNE` instruction is always false. (e.g. The s operand is the zero register and t is a nonzero constant expression.)

Instruction seq: result is always true.

The result of the condition tested by the `SEQ` instruction is always false. (e.g. The s operand is the zero register and t is the constant 0.)

Invalid merge entity size.

The section merge entity size must be non-negative.

Invalid number.

The constant was in an unrecognized format. Check the constant's prefix and radix.

J

Jump address range overflow (0x%lx).

The target address of the jump instruction is outside the 228-byte "page".

L

Left operand is a bignum; integer 0 assumed.

The left operand in the expression is a bignum rather than an integer. The assembler performs expression evaluation on only integers so it assumes integer 0 for this operand.

Left operand is a float; integer 0 assumed.

The left operand in the expression is a float rather than an integer. The assembler performs expression evaluation on only integers so it assumes integer 0 for this operand.

Line numbers must be positive; line number %d rejected.

This directive accepts only positive integers for the line number.

M

Missing close quote; (assumed).

The single-character quote is not properly closed.

Missing operand; zero assumed.

An operand (probably the right-size operand) is missing in the expression. The assembler assumes integer 0 and continues.

O

Operand overflow.

The constant expression used as in the (basereg+offset) operand accepts only 32-bit signed constants.

R

Repeat < 0; .fill ignored.

The repeat argument to the .fill directive must be non-negative.

Right operand is a bignum; integer 0 assumed.

The right operand in the expression is a bignum rather than an integer. The assembler performs expression evaluation on only integers so it assumes integer 0 for this operand.

Right operand is a float; integer 0 assumed.

The right operand in the expression is a float rather than an integer. The assembler performs expression evaluation on only integers so it assumes integer 0 for this operand.

S

Setting incorrect section attributes for *name*

When setting section attributes on a special section, the section's attributes should match those of the predefined type. The assembler uses the predefined type for the section.

Setting incorrect section type for *name*

When setting section attributes on a special section, the section's attributes should match those of the predefined type. The assembler uses the predefined type for the section.

Size negative; .fill ignored.

The size argument to the `.fill` directive must be non-negative.

T

Tried to set unrecognized symbol: *name*

The symbol in the .set directive was not a recognized PIC32 MCU assembler symbol.

Truncated file *filename*, *num1* of *num2* bytes read.

The number of bytes read from the `.incbin` file was fewer than the number specified in the counts argument.

U

Unknown escape *escape* in string; ignored.

The string contains an unrecognized backslash-escaped character. Check that the character following the backslash is correct.

Used $at without .set noat.

This code is using the $at (assembler temporary) register, but the assembler may use it when generating synthesized macro instruction. Use the `.set noat` directive to tell the assembler not to quietly use this register

5.5 MESSAGES

The assembler generates messages when a non-critical assumption is made so that the assembler could continue assembling a flawed program. Messages may be ignored. However, messages can sometimes point out bugs in your program.

Part 2 – MPLAB XC32 Object Linker

NOTES:

Chapter 6. Linker Overview

6.1 INTRODUCTION

MPLAB XC32 Object Linker (xc32-ld) produces binary code from relocatable object code and archives for the PIC32 MCU family of devices. The 32-bit linker is a Windows console application that provides a platform for developing executable code. The linker is a part of the GNU linker from the Free Software Foundation.

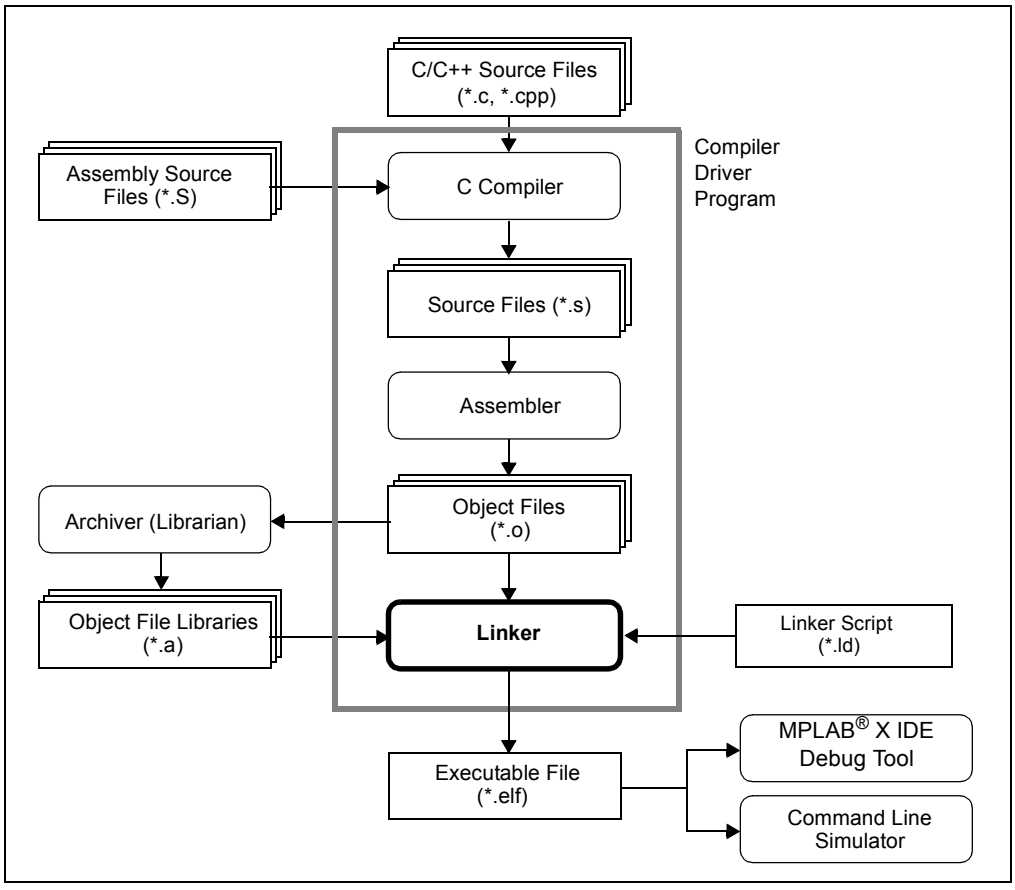
Topics covered in this chapter are:

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

6.2 LINKER AND OTHER DEVELOPMENT TOOLS

The PIC32 linker translates object files from the PIC32 assembler, and archives files from the PIC32 archiver/librarian, into an executable file. See Figure 6-1 for an overview of the tools process flow.

FIGURE 6-1: TOOLS PROCESS FLOW



6.3 FEATURE SET

Notable features of the linker include:

- User-defined minimum stack allocation
- User-defined heap allocation
- Available for Windows
- Linker scripts for all current PIC32 devices
- Command-Line Interface
- Integrated component of MPLAB X IDE

6.4 INPUT/OUTPUT FILES

Linker input and output files are listed below.

TABLE 6-1: LINKER FILES

Extension	Description
Input	
`.o`	Object Files
`.a`	Library Files
`.ld`	Linker Script File
Output	
`.elf, .out`	Linker Output Files
`.map`	Map File

Unlike the Microchip MPLINK™ linker, the 32-bit linker does not generate absolute listing files. The 32-bit linker is capable of creating a map file and a binary ELF file (that may or may not contain debugging information). For text output similar to MPLINK's listing file, run the ELF file through the xc32-objdump binary utility.

6.4.1 Object Files

Relocatable code produced by the assembler. The linker accepts the ELF object file format.

6.4.2 Library Files

A collection of object files grouped together for convenience.

6.4.3 Linker Script File

Linker scripts, or command files:

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

For more on linker script files, see **Chapter 8. "Linker Scripts"**.

EXAMPLE 6-1: LINKER SCRIPT

> **Note:** This simplified linker-script example is for illustrative purposes only; it is not a complete, working, linker script.

```
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH(pic32mx)
ENTRY(_reset)

MEMORY
{
 kseg0_program_mem(rx): ORIGIN=0x9D000000, LENGTH=0x8000
 kseg0_boot_mem       : ORIGIN=0x9FC00490, LENGTH=0x970
 exception_mem        : ORIGIN=0x9FC01000, LENGTH=0x1000
 kseg1_boot_mem       : ORIGIN=0xBFC00000, LENGTH=0x490
 debug_exec_mem       : ORIGIN=0xBFC02000, LENGTH=0xFF0
 config3              : ORIGIN=0xBFC02FF0, LENGTH=0x4
 config2              : ORIGIN=0xBFC02FF4, LENGTH=0x4
 config1              : ORIGIN=0xBFC02FF8, LENGTH=0x4
 config0              : ORIGIN=0xBFC02FFC, LENGTH=0x4
 kseg1_data_mem  (w!x): ORIGIN=0xA0000000, LENGTH=0x2000
 sfrs                 : ORIGIN=0xBF800000, LENGTH=0x100000
}

SECTIONS
{
  .text ORIGIN(kseg0_program_mem)  :
  {
    _text_begin = . ;
    *(.text .stub .text.* )
    *(.mips16.fn.*)
    *(.mips16.call.*)
    _text_end = . ;
  } >kseg0_program_mem =0
  .data   :
  {
    _data_begin = . ;
    *(.data .data.* .gnu.linkonce.d.*)
    KEEP (*(.gnu.linkonce.d.*personality*))
    *(.data1)
  } >kseg1_data_mem AT>kseg0_program_mem
  .bss   :
  {
    *(.dynbss)
    *(.bss .bss.* )
    *(COMMON)
    . = ALIGN(32 / 8) ;
  } >kseg1_data_mem
  .stack ALIGN(4) :
  {
    . += _min_stack_size ;
  } >kseg1_data_mem
}
```

6.4.4 Linker Output Files

By default, the name of the linker output binary file is `a.out`. You can override the default name by specifying the `-o` option on the command line. The MPLAB X IDE project manager uses the `-o` option to name the output file *`projectname`*`.elf`, where *`projectname`* is the name of your MPLAB X IDE project.

The format of the binary file is an Executable and Linking Format (ELF) file. The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). The ELF specification is the result of the work of the Tool Interface Standards (TIS) Committee, an association of members of the microcomputer industry formed to work toward standardization of the software interfaces visible to development tools.

The debugging information within the ELF file is in the DWARF Debugging Information format. Also a result of the work of the TIS Committee, the DWARF format uses a series of debugging entries to define a low-level representation of a source program. A DWARF consumer, such as MPLAB X IDE, can then use the representation to create an accurate picture of the original source program

6.4.5 Map File

The map files produced by the linker consist of:

- Archive Member Table – lists the name of any members from archive files that are included in the link.
- Memory Usage Report – shows the starting address and length of all output sections in program memory and data memory. It also shows a percent utilization of memory in the region.
- Memory Configuration – lists all of the memory regions defined for the link.
- Linker Script and Memory Map – shows modules, sections and symbols that are included in the link as specified in the linker script.
- Outside Cross Reference Table (optional) - shows symbols, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files listed contain references to the symbol.

EXAMPLE 6-2: MAP FILE

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

size\libc.a(general-exception.o)
         size/crt0.o (_general_exception_context)
size\libc.a(default-general-exception-handler.o)
         size\libc.a(general-exception.o) (_general_exception_handler)
size\libc.a(default-bootstrap-exception-handler.o)
         size/crt0.o (_bootstrap_exception_handler)
size\libc.a(default-on-reset.o)
         size/crt0.o (_on_reset)
size\libc.a(default-on-bootstrap.o)
         size/crt0.o (_on_bootstrap)
size\libc.a(default-nmi-handler.o)
         size/crt0.o (_nmi_handler)

Microchip PIC32 Memory-Usage Report

kseg0 Program-Memory Usage
section    address  length   (dec)  Description
------- ----------  --------------  -----------
.text   0x9d000000   0x678    1656  Application's executable code
.rodata 0x9d000678    0x14      20  Read-only constant data
.data   0x9d00068c     0xf     244  Data-initialization template
.sdata  0x9d000780     0x4       4  Small data-initialization template
Total kseg0_program_mem used:
                     0x784    1924  0.4% of 0x80000

kseg0 Boot-Memory Usage
section  address    length   (dec)  Description
-------  ---------- --------------  -----------
.startup 0x9fc00490  0x1e0     480  C startup code
Total kseg0_boot_mem used:
                     0x1e0     480  19.9% of 0x970

Exception-Memory Usage
section    address    length   (dec)  Description
-------    ---------- --------------  -----------
.app_excpt 0x9fc01180   0x10      16  General-Exception handler
.vector_1  0x9fc01220    0x8       8  Interrupt Vector 1
Total exception_mem used  :
                        0x18      24  0.6% of 0x1000

kseg1 Boot-Memory Usage
section    address     length   (dec)  Description
-------    ----------  --------------  -----------
.reset     0xbfc00000    0x10      16  Reset handler
.bev_excpt 0xbfc00380    0x10      16  BEV-Exception handler
Total kseg1_boot_mem used  :
                         0x20      32  2.7% of 0x490
        ---------------------------------------------
         Total Program Memory used  :
                         0x99c    2460  0.5% of 0x81e00
        ---------------------------------------------
```

```
kseg1 Data-Memory Usage
section    address     length   (dec)  Description
-------    ----------  --------------  -----------
.data      0xa0000000    0xf4     244  Initialized data
.sdata     0xa00000f4     0x4       4  Small initialized data
.sbss      0xa00000f8     0x4       4  Small uninitialized data
.bss       0xa00000fc   0x10c     268  Uninitialized data
.heap      0xa0000208   0x800    2048  Dynamic Memory heap
.stack     0xa0000a08   0x400    1024  Min space reserved for stack
         Total kseg1_data_mem used  :
                      0xe08    3592  11.0% of 0x8000
        ------------------------------------------------------------
            Total Data Memory used  :
                      0xe08     3592  11.0% of 0x8000
        ------------------------------------------------------------
Memory Configuration

Name             Origin             Length             Attributes
kseg0_program_mem 0x9d000000         0x00080000         xr
kseg0_boot_mem   0x9fc00490         0x00000970
exception_mem    0x9fc01000         0x00001000
kseg1_boot_mem   0xbfc00000         0x00000490
config0          0xbfc02ffc         0x00000004
kseg1_data_mem   0xa0000000         0x00008000         w !x
sfrs             0xbf800000         0x00100000
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD size/crt0.o
                0x00000800       _min_heap_size = 0x800
START GROUP
LOAD size\libc.a
LOAD size\libm.a
LOAD size\libmchp_peripheral_32MX360F512L.a
END GROUP
LOAD C:/xc32-Tools/bin/../lib/gcc/pic32mx/3.4.4/size\libgcc.a
              0x00000400    PROVIDE (_min_stack_size, 0x400)
              0x00000000    PROVIDE (_min_heap_size, 0x0)
LOAD ./proc/32MX360F512L\processor.o
              0x00000001    PROVIDE (_vector_spacing, 0x1)
              0x9fc01000    _ebase_address = 0x9fc01000
              0xbfc00000    _RESET_ADDR = 0xbfc00000
              0xbfc00380    _BEV_EXCPT_ADDR = 0xbfc00380
              0x9fc01180    _GEN_EXCPT_ADDR = (_ebase_address + 0x180)

.reset        0xbfc00000    0x10
 *(.reset)
 .reset       0xbfc00000    0x10 size/crt0.o
              0xbfc00000        _reset
.bev_excpt    0xbfc00380    0x10
 *(.bev_handler)
 .bev_handler 0xbfc00380    0x10 size/crt0.o
.vector_0     0x9fc01200    0x0
 *(.vector_0)

.startup      0x9fc00490    0x1e0
 *(.startup)
 .startup     0x9fc00490    0x1e0 size/crt0.o
```

```
.text          0x9d000000    0x678
               0x9d000000          _text_begin = .
 *(.text .stub .text.* .gnu.linkonce.t.*)
 .text         0x9d000000    0x18 size/crt0.o
 .text         0x9d000018    0x110 intermediate\object.o
               0x9d000089          testfunct
               0x9d0000a0          main
               0x9d000018          foo
 .text         0x9d000128    0xc intermediate   est.o
               0x9d000128          mylabel
 .text.general_exception
               0x9d000134    0xd0 size\libc.a(general-exception.o)
               0x9d000134          _general_exception_context
 .text._general_exception_handler
       0x9d0005bc 0x8 size\libc.a(default-general-exception-handler.o)
             0x9d0005bc          _general_exception_handler
 .text._bootstrap_exception_handler
    0x9d0005c4  0x8 size\libc.a(default-bootstrap-exception-handler.o)
               0x9d0005c4          _bootstrap_exception_handler
 .text._on_reset
               0x9d0005cc    0x8 size\libc.a(default-on-reset.o)
               0x9d0005cc          _on_reset
 .text._on_bootstrap
               0x9d0005d4    0x8 size\libc.a(default-on-bootstrap.o)
               0x9d0005d4          _on_bootstrap
 .text         0x9d0005dc    0x18 size\libc.a(default-nmi-handler.o)
               0x9d0005dc          _nmi_handler
.sdata         0xa00000f4    0x4 load address 0x9d000780
               0xa00000f4          _sdata_begin = .
.heap          0xa0000208    0x800
               0xa0000208          _heap = .
               0xa0000a08          . = (. + _min_heap_size)
 *fill*        0xa0000208    0x800 00

.stack         0xa0000a08    0x400
               0xa0000e08          . = (. + _min_stack_size)
 *fill*        0xa0000a08    0x400 00

.ramfunc       0xa0001000    0x0 load address 0x9d000784
               0xa0001000          _ramfunc_begin = .
 *(.ramfunc .ramfunc.*)
               0xa0001000          . = ALIGN (0x4)
               0xa0008000          _stack =
                                  (_ramfunc_length >0x0)?
                                  (_ramfunc_begin - 0x4):0xa0008000

OUTPUT(test-2.elf elf32-tradlittlemips)

Cross Reference Table

Symbol          File
PORTE           ./proc/32MX360F512L\processor.o
                size\libc.a(default-nmi-handler.o)
                size\libc.a(general-exception.o)
                intermediate/test.o
                size/crt0.o
foo             intermediate\cobject.o
main            intermediate\cobject.o
                size/crt0.o
mylabel         intermediate\asmobject.o
funct           intermediate\cobject.o
```

NOTES:

Chapter 7. Linker Command-Line Interface

7.1 INTRODUCTION

MPLAB XC32 Object Linker (xc32-ld) may be used on the command line interface as well as with MPLAB X IDE.

Topics covered in this chapter are:

- Linker Interface Syntax
- Compilation-Driver Linker Interface Syntax
- Options that Control Output File Creation
- Options that Control Run-time Initialization
- Options that Control Informational Output
- Options that Modify the Link Map Output

7.2 LINKER INTERFACE SYNTAX

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

```
xc32-ld [options] file...
```

> **Note:** command-line options are case sensitive.

For instance, a frequent use of `xc32-ld` is to link object files and archives to produce a binary file. To link a file `hello.o`:

```
xc32-ld -o output hello.o -lpic32
```

This tells `xc32-ld` to produce a file called `output` as the result of linking the file `hello.o` with the archive `libpic32.a`.

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

```
xc32-ld -o output hello.o --start-group -lpic32 -lm -lc --end-group
```

The command-line options to `xc32-ld` may be specified in any order, and may be repeated at will. Repeating most options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option. Options that may be meaningfully specified more than once are noted in the descriptions below.

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

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

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

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

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

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

7.3 COMPILATION-DRIVER LINKER INTERFACE SYNTAX

In practice, the linker is usually invoked via xc32-gcc, the compilation driver. The basic form of the compilation-driver command line is:

```
xc32-gcc [options] files
```

> **Note:** Command-line options and filename extensions are case sensitive.

To pass a linker option from the compilation driver to the linker, use the `-Wl,`*`option`* option.

EXAMPLE 7-1: COMPILATION-DRIVER COMMAND LINE

```
xc32-gcc -mprocessor=32MX360F512L "input.o" -o"output.elf"
   -Os -Wl,--defsym=_min_heap_size=2048,-Map="mapfile.map",
   --cref,--report-mem
```

Calling the linker via the compilation driver has a few advantages over calling the linker directly.

- The driver's `-mprocessor` option allows the driver to pass the correct device-specific include-file and library search paths to the linker. For instance, when specifying `-mprocessor=32MX360F512L`, the driver passes the corresponding device-specific library search path, `pic32mx/lib/proc/32MX360F512L`, to the linker. This path allows the linker to find the correct default linker script and processor library for the target device.
- The driver accepts the C compiler's optimization, ISA mode, and floating-point support options required to select the appropriate multilib permutation. For example, when passing the `-Os` size optimization option, the driver passes `pic32mx/lib/size` as a library search path so that the linker uses the pre-compiled libraries optimized for size. See the "*MPLAB XC32 C/C++ Compiler User's Guide"* (DS51686) for more information on the C compiler's multilib feature.

7.4 OPTIONS THAT CONTROL OUTPUT FILE CREATION

Output file creation options are:

- `-( archives -), --start-group archives, --end-group`
- `-d, -dc, -dp`
- `--defsym sym=expr`
- `--discard-all (-x)`
- `--discard-locals (-X)`
- `--fill=option`
- `--gc-sections`
- `--library name (-l name)`
- `--library-path <dir> (-L <dir>)`
- `-nodefaultlibs`
- `-nostartfiles`
- `-nostdlib`
- `--output file (-o file)`
- `--p PROC`
- `--relocatable (-r, -i, -Ur)`
- `--retain-symbols-file file`
- `--section-start sectionname=org`
- `--script file (-T file)`
- `--strip-all (-s)`
- `--strip-debug (-S)`
- `-Tbss address`
- `-Tdata address`
- `-Ttext address`
- `--undefined symbol (-u symbol)`
- `--no-undefined`
- `--wrap symbol`

7.4.1 `-( archives -), --start-group archives, --end-group`

Start and end a group.

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

7.4.2 `-d, -dc, -dp`

Force common symbols to be defined.

Assign space to common symbols even if a relocatable output file is specified (with `-r`). The script command `FORCE_COMMON_ALLOCATION` has the same effect.

7.4.3 --defsym *sym=expr*

Define a symbol.

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

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

7.4.4 --discard-all (-x)

Discard all local symbols.

7.4.5 --discard-locals (-X)

Discard temporary local symbols.

7.4.6 --fill=option

--fill=option

Fill unused program memory. The format is:

--fill=[wn:]expression[@address[:end_address] | unused]

address and end_address will specify the range of program memory addresses to fill. If end_address is not provided, then the expression will be written to the specific memory location at address address. The optional literal value unused may be specified to indicate that all unused memory will be filled. If none of the location parameters are provided, all unused memory will be filled. expression will describe how to fill the specified memory. The following options are available:

Single value:

xc32-ld --fill=0x12345678@unused

Range of values:

xc32-ld --fill=1,2,3,4,097@0x9d000650:0x9d000750

Incrementing value:

xc32-ld --fill=7+=711@unused

By default, the linker will fill using data that is instruction-word length. For 32-bit devices, the default fill width is 32 bits. However, you may specify the value width using [wn:], where n is the fill value's width and n belongs to [1, 2, 4, 8]. Multiple fill options may be specified on the command line; the linker will always process fill options at specific locations first.

7.4.7 --gc-sections

Enable garbage collection of unused input sections. This option is not compatible with -r. The default behavior (of not performing this garbage collection) can be restored by specifying *--no-gc-sections* on the command line.

When link-time garbage collection is in use, marking sections that should not be eliminated is often useful. Mark the section by surrounding an input section's wildcard entry with KEEP(), as in KEEP(*(.init)) or KEEP(SORT_BY_NAME(*)(.ctors)).

7.4.8 --library *name* (-l *name*)

Search for library *name*.

Add archive file *name* to the list of files to link. This option may be used any number of times. xc32-ld will search its path-list for occurrences of lib*name*.a for every *name* specified. The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol that was undefined in some object that appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again. See the -(option for a way to force the linker to search archives multiple times. You may list the same archive multiple times on the command line.

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

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

Add <*dir*> to library search path.

Add path <*dir*> to the list of paths that xc32-ld will search for archive libraries and xc32-ld control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. All -L options apply to all -l options, regardless of the order in which the options appear. The library paths can also be specified in a link script with the SEARCH_DIR command. Directories specified this way are searched at the point in which the linker script appears in the command line.

7.4.10 -nodefaultlibs

Do not use the standard system libraries when linking. Only the libraries you specify are passed to the linker. The compiler may generate calls to memcmp, memset and memcpy. These entries are usually resolved by entries in the standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.

7.4.11 -nostartfiles

Do not pass the default prebuilt C startup file (pic32mx/lib/crt0.o) to the linker. You will provide your own version of the startup code for the application.

7.4.12 -nostdlib

Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify are passed to the linker. The compiler may generate calls to memcmp, memset and memcpy. These entries are usually resolved by entries in standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.

7.4.13 --output *file* (-o *file*)

Set output ELF file name.

Use *file* as the name for the program produced by xc32-ld; if this option is not specified, the name a.out is used by default.

7.4.14 --p PROC

Specify the target processor (e.g., 32MX795F512L).

Specify a target processor for the link.

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

Generate relocatable output.

I.e., generate an output file that can in turn serve as input to xc32-ld. This is often called partial linking. If this option is not specified, an absolute file is produced.

7.4.16 --retain-symbols-file *file*

Keep only symbols listed in *file*.

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

7.4.17 --section-start sectionname=***org***

Locate a section in the output file at the absolute address given by *org*. You may use this option as many times as necessary to locate multiple sections in the command line. *org* must be a single hexadecimal integer; for compatibility with other linkers, you may omit the leading '0x' that is usually associated with hexadecimal values.

Note: There should be no white space between sectionname, the equals sign (=), and *org*.

7.4.18 --script *file* (-T *file*)

Read linker script.

Read link commands from the file *file*. These commands replace xc32-ld's default link script (rather than adding to it), so *file* must specify everything necessary to describe the target format. If *file* does not exist, xc32-ld looks for it in the directories specified by any preceding -L options. Multiple -T options accumulate.

7.4.19 --strip-all (-s)

Strip all symbols. Omit all symbol information from the output file.

7.4.20 --strip-debug (-S)

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

7.4.21 -Tbss *address*

Set address of .bss section.

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

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

7.4.22 -Tdata *address*

Set address of .data section.

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

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

7.4.23 -Ttext *address*

Set address of .text section.

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

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

7.4.24 --undefined *symbol* (-u *symbol*)

Start with undefined reference to *symbol*.

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

7.4.25 --no-undefined

Allow no undefined symbols.

7.4.26 --wrap *symbol*

Use wrapper functions for *symbol*

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

Here is a trivial example:

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

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

7.5 OPTIONS THAT CONTROL RUN-TIME INITIALIZATION

Run-time initialization options are:

- `--data-init`
- `--no-data-init`
- `--defsym=_min_stack_size=size`
- `--defsym=_min_heap_size=size`

7.5.1 --data-init

Support initialized data. (This is the default.)

Create a special output section named `.dinit` as a template for the run-time initialization of data. The C start-up module in libpic32.a interprets this template and copies initial data values into initialized data sections. Other data sections (such as `.bss`) are cleared before the main() function is called. Note that the persistent data section (`.pbss`) is not affected by this option.

The

7.5.2 --no-data-init

Do not support initialized data.

Suppress the template which is normally created to support run-time initialization of data. When this option is specified, the linker will select a shorter form of the C start-up module in libpic32.a. If the application includes data sections which require initialization, a warning message will be generated and the initial data values discarded. Storage for the data sections will be allocated as usual.

7.5.3 --defsym=_min_stack_size=*size*

The default linker script provides a minimum stack size of 1024 bytes. Use the `--defsym` option to define the `_min_stack_size` symbol to change this default *`size`* value. Note that the actual effective stack size may be larger than the minimum size.

```
xc32-gcc foo.c -Wl,--defsym=_min_stack_size=1536
```

7.5.4 --defsym=_min_heap_size=*size*

The default linker script provides a heap size of 0 bytes. Use the --defsym option to define the _min_heap_size symbol to change this default size value. The linker creates a heap with a size defined by this value.

```
xc32-gcc foo.c -Wl,--defsym=_min_heap_size=2048
```

7.6 OPTIONS THAT CONTROL MULTILIB LIBRARY SELECTION

Multilibs are a set of prebuilt target libraries. Each target library in the multilib gets built with a different set of compiler options. Multilibs provide the linker with the capability to match a target library with the compiler options used to build an application. The pre-built target libraries represent the most common combinations of compiler options.

When the compilation driver is called to link an application, the driver chooses the version of the target library that corresponds to the application options. These options should be passed to the compilation driver, not the linker proper. The compilation driver then translates the options to the appropriate `-L` library search path when calling the linker.Size versus speed (-Os vs. -O3)

Select either `-Os` to optimize for size or `-O0` through `-O3` to optimize for speed.

-O0

Select the unoptimized multilib target-library permutation. (This is the default for the command-line interface, but MPLAB X IDE's project manager may pass one of the other optimization options by default.)

-O1

Select the multilib target-library permutation built with optimization level 1.

-O2

Select the multilib target-library permutation built with optimization level 2. This optimization level provides a good balance between execution speed and code size. This multilib optimization level is a good choice for most applications.

-O3

Select the multilib target-library permutation built with optimization level 3. This optimization maximizes execution speed.

-Os

Select the multilib target-library permutation built optimized for code size, for example:

```
xc32-gcc foo.o -Os -o project.elf
```

7.6.1 Instruction Set Mode (MIPS32/MIPS16E/microMIPS

Selects multilib permutation based on -mips16 or -mmicromips or -mips32r2, for example:

```
xc32-gcc foo.o -O3 -mips16 -o project.elf
```

7.6.2 Software floating-point versus no floating-point support

The no-float library permutations typically have significantly less overhead than the floating-point library permutations. If your application does not require floating-point support, use this option.

-mno-float

Selects the multilib target-library permutation that does not support software floating-point operations.

Example:

```
xc32-gcc foo.o -Os -mno-mips16 -mno-float -o project.elf
```

-msoft-float

Selects the multilib-target-library permutation with full software floating-point support.

7.7 OPTIONS THAT CONTROL INFORMATIONAL OUTPUT

Information output options are:

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

7.7.1 `--check-sections`

Check section addresses for overlaps. **(This is the default.)** Normally the linker will perform this check, and it will produce a suitable error message if it finds an overlap.

7.7.2 `--no-check-sections`

Do not check section addresses for overlaps. Use for diagnosing memory allocation issues only.

7.7.3 `--help`

Print option help.

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

7.7.4 `--no-warn-mismatch`

Do not warn about mismatched input files.

Normally `xc32-ld` will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells `xc32-ld` that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures that the linker errors are inappropriate.

7.7.5 `--report-mem`

Print a memory usage report.

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

7.7.6 `--trace (-t)`

Trace file.

Print the names of the input files as `xc32-ld` processes them.

7.7.7 --trace-symbol *symbol* (-y *symbol*)

Trace mentions of *symbol*.

Print the name of each linked file in which *symbol* appears. This option may be given any number of times. This option is useful when you have an undefined symbol in your link but do not know where the reference is coming from.

7.7.8 -V

Print version and other information.

7.7.9 --verbose

Output lots of information during link.

Display the version number for xc32-ld. Display the input files that can and cannot be opened. Display the linker script if using a default built-in script.

7.7.10 --version (-v)

Print version information.

7.7.11 --warn-common

Warn about duplicate common symbols.

Warn when a common symbol is combined with another common symbol or with a symbol definition. This option allows you to find potential problems from combining global symbols. There are three kinds of global symbols, illustrated here by C examples:

```
int i = 1;
```

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

```
extern int i;
```

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

```
int i;
```

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

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

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

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

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

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

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

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

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

Merging a common symbol with a previous larger common symbol.

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

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

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

7.7.12 --warn-once

Warn only once for each undefined symbol, rather than once per module that refers to it.

7.7.13 --warn-section-align

Note that section-alignment gaps are normal. This option helps you identify ways to minimize gaps.

Warn if start of section changes due to alignment. This means a gap has been introduced into the (normally sequential) allocation of memory. Typically, an input section will set the alignment. The address will only be changed if it is not explicitly specified; that is, if the SECTIONS command does not specify a start address for the section.

7.8 OPTIONS THAT MODIFY THE LINK MAP OUTPUT

Link map output modifying options are:

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

7.8.1 `--cref`

Output cross reference table.

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

7.8.2 `--print-map (-M)`

Print map file on standard output. A link map provides information about the link, including the following:

- Where object files and symbols are mapped into memory.
- How common symbols are allocated.
- All archive members included in the link, with a mention of the symbol which caused the archive member to be brought in.

7.8.3 `-Map` *`file`*

Write a map file.

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

Chapter 8. Linker Scripts

8.1 INTRODUCTION

Linker scripts are used to control MPLAB XC32 Object Linker (xc32-ld) functions. By default, the linker uses a built-in linker script with a device-specific include file. However, you can also customize your linker script for specialized control of the linker in your application.

Topics covered in this chapter are:

- Overview of Linker Scripts
- Command Line Information
- Default Linker Script
- Adding a Custom Linker Script to an MPLAB X IDE Project
- Linker Script Command Language
- Expressions in Linker Scripts

8.2 OVERVIEW OF LINKER SCRIPTS

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

- allocation of data memory and program memory
- mapping of sections from input files into the output file
- construction of special data structures (such as interrupt vector tables)

Linker scripts are text files that contain a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol.

8.3 COMMAND LINE INFORMATION

Linker scripts are specified on the command line using either the `-T` option or the `--script` option (see **Section 7.4 "Options that Control Output File Creation"**):

```
xc32-ld -o output.elf input.o --script mylinkerscript.ld
```

If the linker is invoked through xc32-gcc, add the `-Wl,` prefix to allow the option to be passed to the linker:

```
xc32-gcc -o output.elf input.o -Wl,--script,mylinkerscript.ld
```

8.4 DEFAULT LINKER SCRIPT

For PIC32MX Devices Only:

If no linker script is specified on the command line, the linker will use an internal version known as the built-in default linker script. The default linker script has section mapping that is appropriate for all PIC32 MCUs. It uses an `INCLUDE` directive to include the device-specific memory regions.

The default linker script is appropriate for most PIC32 MCU applications. Only applications with specific memory-allocation needs will require an application-specific linker script. The default linker script can be examined by invoking the linker with the `--verbose` option:

```
xc32-ld --verbose
```

In a normal tool-suite installation, a copy of the default linker script is located at `\pic32mx\lib\ldscripts\elf32pic32mx.x`. Note that this file is only a copy of the default linker script. The script that the linker uses is internal to the linker.

The device-specific portion of the linker script is located in `\pic32mx\lib\proc\`*`device`*`\procdefs.ld`, where *`device`* is the device value specified to the `-mprocessor` compilation-driver (xc32-gcc) option.

For PIC32MZ and Later Devices:

Single-file linker script for PIC32MZ and later devices: The linker script for PIC32MZ devices are contained within a single file (e.g. pic32mx/lib/proc/32MZ2048ECH100/p32MZ2048ECH100.ld). This eliminates the dependency on two files (elf32pic32mx.x and procdefs.ld) used by the older linker-script model. Like before, the xc32-gcc compilation driver will pass the device-specific linker script to the linker when building with -mprocessor=device option.

> **Note:** The "*MPLAB XC32 C/C++ Compiler User's Guide*" (DS51686) examines the contents of the default linker script in detail. The discussion applies to both assembly-code and C-code projects.

The default linker script maps each standard input section to one or more specific MEMORY regions. In turn, each MEMORY region maps to an address segment on the PIC32 MCU (e.g. kseg0, kseg1). See the Section 3 of the *PIC32MX Family Reference Manual* (DS61115) for a full description of the user/kernel address segments.

The table below shows how the default linker script maps standard sections to MEMORY regions.

TABLE 8-1: PIC32 RESERVED, STANDARD SECTION NAMES IN DEFAULT LINKER SCRIPT

Section Name	Generated by	Final Location	Default linker-script MEMORY region
.reset	Reset handler	Executable boot-code segment	kseg0_boot_mem
.bev_excpt	BEV-Exception handler	Executable boot-code segment	kseg0_boot_mem
.app_excpt	General-Exception handler	Executable boot-code segment	kseg0_boot_mem
.vector_n	Interrupt Vector n	Executable boot-code segment	kseg0_boot_mem
.startup	C startup code	Executable boot-code segment	kseg0_boot_mem
.text	Compiler- or assembler-generated instructions	Executable code segment	kseg0_program_mem
.rodata	Strings and C data declared const	Read-only data segment	kseg0_program_mem
.sdata2	Small initialized constant global and static data	Read-only data segment	kseg0_program_mem
.sbss2	Uninitialized constant global and static data (i.e., variables which will always be zero)	Read-only data segment	kseg0_program_mem
.data	Variables >n bytes (compiled -Gn) with an initial value. Values copied from program memory to data memory at C startup.	Initialized data segment	kseg1_data_mem & kseg0_program_mem
.sdata	Variables <=n bytes (compiled -Gn) with an initial value. Used for gp-relative addressing.	Small initialized data segment	kseg1_data_mem & kseg0_program_mem
.lit4 / .lit8	Constants (usually floating point) which the assembler decides to store in memory rather than in the instruction stream. Used for gp-relative addressing.	Small initialized data segment	kseg1_data_mem & kseg0_program_mem
.sbss	Uninitialized variables <=n bytes (compiled -Gn). Used for gp-relative addressing.	Small zero-filled segment	kseg1_data_mem
.bss	Uninitialized larger variables	Zero-filled segment	kseg1_data_mem
.heap	Heap used for dynamic memory	Reserved by linker script	kseg1_data_mem
.stack	Minimum space reserved for stack	Reserved by linker script	kseg1_data_mem
.ramfunc	RAM-functions, copied from program memory to data memory at C startup	Initialized data segment	kseg1_data_mem & kseg0_program_mem
.reginfo .stab* .debug*	Debug information	Not in load image	n/a
.line	DWARF debug information	Not in load image	n/a
.comment	#ident/.ident strings	Not in load image	n/a

Note: The table above contains sections that are no longer mapped in the linker script. Starting with XC32 v2.00, the best-fit allocator allocates them.

8.5 ADDING A CUSTOM LINKER SCRIPT TO AN MPLAB X IDE PROJECT

The standard default 32-bit linker scripts are general purpose and will satisfy the demands of most applications. However, an occasion may arise where a custom linker script is required.

Copy the default linker-script file (e.g., `pic32mx/lib/proc/32MZ2048ECH100/p32MZ2048ECH100.ld`) in your application's project directory. Add the new `*.ld` file to your project. It should now appear in the project tree under “Linker Files”.

Customizations that you make to your new `*.ld` file should now affect your project.

You may wish to retain unused sections in a custom linker script, since unused sections will not impact application memory usage. If a section must be removed for a custom script, C style comments can be used to disable it.

8.6 LINKER SCRIPT COMMAND LANGUAGE

Linker scripts are text files that contain a series of commands. Each command is either a keyword, possibly followed by arguments, or an assignment to a symbol. Multiple commands may be separated using semicolons. White space is generally ignored, but there are some cases where white space is significant. For instance, white space is required around operators.

Strings such as file or format names can normally be entered directly. If the file name contains a character such as a comma which would otherwise serve to separate file names, the file name may be specified in double quotes. There is no way to use a double quote character in a file name.

Comments may be included just as in C, delimited by `/*` and `*/`. As in C, comments are syntactically equivalent to white space.

8.6.1 Basic Linker Script Concepts

The linker combines input files into a single output file. The output file and each input file are in a special data format known as an ELF object file format. Each file is called an object file. Each object file has, among other things, a list of sections. A section in an input file is called an input section; similarly, a section in the output file is an output section.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the section contents. A section may be marked as loadable, which mean that the contents should be loaded into memory when the output file is run. A section with no contents may be allocatable, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out).

Every loadable or allocatable output section has two addresses. The first is the VMA, or virtual memory address. This is the address the section will have when the output file is run. The second is the LMA, or load memory address. This is the address at which the section will be loaded. In most cases, the two addresses will be the same. An example of when they might be different is when a section is intended to contain RAM-located functions (e.g. the default `.ramfunc` section). In this case, the program-memory address would be the LMA and the data-memory address would be the VMA.

> **Note:** Both the VMA and the LMA use the PIC32 MCU's virtual address. See the *PIC32MX Family Reference Manual* (DS61115) for a description of the PIC32MX Virtual-to-Physical Fixed Memory Mapping. In addition, the family reference manual describes the PIC32 memory layout.

The sections in an object file can be viewed by using the `xc32-objdump` program with the `-h` option.

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If a C program is compiled into an object file, a defined symbol will be created for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

Symbols in an object file can be viewed by using the `xc32-nm` program, or by using the `xc32-objdump` program with the `-t` option.

8.6.2 Commands Dealing with Files

Several linker script commands deal with files.

INCLUDE filename

Include the linker script filename at this point. The file will be searched for in the current directory, and in any directory specified with the `-L` option. Calls to `INCLUDE` may be nested up to 10 levels deep.

```
INPUT(file, file, ...)
INPUT(file file ...)
```

The `INPUT` command directs the linker to include the named files in the link, as though they were named on the command line. The linker will first try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of `-L` in **Section 7.4.9 "--library-path <dir> (-L <dir>)"**.

If `INPUT (-lfile)` is used, `xc32-ld` will transform the name to `libfile.a`, as with the command line argument `-l`.

When the `INPUT` command appears in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

```
GROUP(file, file, ...)
GROUP(file file ...)
```

The `GROUP` command is like `INPUT`, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of archives in **Section 7.4.1 "-(archives -), --start-group archives, --end-group"**.

```
OPTIONAL(file, file, ...)
OPTIONAL(file file ...)
```

The `OPTIONAL` command is analogous to the `INPUT` command, except that the named files are not required for the link to succeed. This is particularly useful for specifying archives (or libraries) that may or may not be installed with the compiler. The default linker scripts provided with the XC32 compiler use the OPTIONAL directive to link the device-specific peripheral libraries.

OUTPUT(filename)

The `OUTPUT` command names the output file. Using `OUTPUT(filename)` in the linker script is exactly like using `-o filename` on the command line (see **Section 7.4.13 "--output file (-o file)"**). If both are used, the command line option takes precedence.

SEARCH_DIR(path)

The `SEARCH_DIR` command adds path to the list of paths where the linker looks for archive libraries. Using `SEARCH_DIR(path)` is exactly like using `-L path` on the command line (see **Section 7.4.9 "--library-path <dir> (-L <dir>)"**). If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

STARTUP(filename)

The `STARTUP` command is just like the `INPUT` command, except that filename will become the first input file to be linked, as though it were specified first on the command line.

8.6.3 Assigning Values to Symbols

A value may be assigned to a symbol in a linker script. This will define the symbol as a global symbol.

8.6.3.1 SIMPLE ASSIGNMENTS

A symbol may be assigned using any of the C assignment operators:

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

The first case will define symbol to the value of expression. In the other cases, symbol must already be defined, and the value will be adjusted accordingly.

The special symbol name '.' indicates the location counter. This symbol may be only used within a `SECTIONS` command.

The semicolon after expression is required.

Expressions are defined in **Section 8.7 "Expressions in Linker Scripts"**.

Symbol assignments may appear as commands in their own right, or as statements within a `SECTIONS` command, or as part of an output section description in a `SECTIONS` command.

The section of the symbol will be set from the section of the expression; for more information, see **Section 8.7.6 "The Section of an Expression"**.

Here is an example showing the three different places that symbol assignments may be used:

```
floating_point = 0;
SECTIONS
{
  .text ORIGIN(kseg0_program_mem)  :
  {
    _text_begin = . ;
    *(.text .stub .text.* )
    _text_end = . ;
  } >kseg0_program_mem =0
   _bdata = (. + 3) & ~ 3;
  .data : { *(.data) }
}
```

In this example, the symbol `floating_point` will be defined as zero. The symbol `_text_end` will be defined as the address following the last `.text` input section. The symbol `_bdata` will be defined as the address following the `.text` output section aligned upward to a 4-byte boundary.

8.6.3.2 PROVIDE

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in the link. For example, traditional linkers defined the symbol `etext`. However, ANSI C requires that `etext` may be used as a function name without encountering an error. The `PROVIDE` keyword may be used to define a symbol, such as `etext`, only if it is referenced but not defined. The syntax is `PROVIDE(symbol = expression)`.

Here is an example of using `PROVIDE` to define `etext`:

```
    SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

The PIC32 default linker script uses the `PROVIDE` command to define the default `_min_stack_size`, `_min_heap_size`, and `_vector_spacing` symbol values.

```
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
PROVIDE(_vector_spacing = 0x00000001);
```

8.6.4 MEMORY Command

The linker's default configuration permits allocation of all available memory. This can be overridden by using the `MEMORY` command.

The `MEMORY` command describes the location and size of blocks of memory in the target. It can be used to describe which memory regions may be used by the linker and which memory regions it must avoid. Sections may then be assigned to particular memory regions. The linker will set section addresses based on the memory regions and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

The syntax of the `MEMORY` command is:

```
 MEMORY
{
  name [(attr)] : ORIGIN = origin, LENGTH = len
  ...
}
```

The name is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names or section names. Each memory region must have a distinct name.

The *attr* string must consist only of the following characters:

`R`	Read-only section
`W`	Read/write section
`X`	Executable section
`A`	Allocatable section
`I`	Initialized section
`L`	Same as `I`
`!`	Invert the sense of any of the following attributes

If an unmapped section matches any of the listed attributes other than `!`, it will be placed in the memory region. The `!` attribute reverses this test, so that an unmapped section will be placed in the memory region only if it does not match any of the listed attributes.

The origin is an expression for the start address of the memory region. The expression must evaluate to a constant before memory allocation is performed, which means that section relative symbols may not be used. The keyword `ORIGIN` may be abbreviated to `org` or `o` (but not, for example, `ORG`).

The `len` is an expression for the size in bytes of the memory region. As with the origin expression, the expression must evaluate to a constant before memory allocation is performed. The keyword `LENGTH` may be abbreviated to `len` or `l`.

In the following example, we specify that there are two memory regions available for allocation: one starting at `0` for 48 kilobytes, and the other starting at `0x800` for two kilobytes. The linker will place into the `rom` memory region every section which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the `ram` memory region.

```
 MEMORY
{
  rom (rx)  : ORIGIN = 0, LENGTH = 48K
  ram (!rx) : org = 0x800, l = 2K
}
```

Once a memory region is defined, the linker can be directed to place specific output sections into that memory region by using the `>region` output section attribute. For example, to specify a memory region named `mem`, use `>mem` in the output section definition. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message.

8.6.5 SECTIONS Command

The SECTIONS command tells the linker how to map input sections into output sections and how to place the output sections in memory.

The format of the SECTIONS command is:

```
   SECTIONS
{
  sections-command
  sections-command
  ...
}
```

Each SECTIONS command may be one of the following:

- an ENTRY command (see **Section 8.6.6 "Other Linker Script Commands"**)
- a symbol assignment (see **Section 8.6.3 "Assigning Values to Symbols"**)
- an output section description
- an overlay description

The ENTRY command and symbol assignments are permitted inside the SECTIONS command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because those commands can be used at meaningful points in the layout of the output file.

Output section descriptions and overlay descriptions are described below.

If a SECTIONS command does not appear in the linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

8.6.5.1 INPUT SECTION DESCRIPTION

The most common output section command is an input section description.

The input section description is the most basic linker script operation. Output sections tell the linker how to lay out the program in memory. Input section descriptions tell the linker how to map the input files into the memory layout.

An input section description consists of a file name optionally followed by a list of section names in parentheses.

The file name and the section name may be wildcard patterns, which are described further below.

The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input .text sections, one would write:

```
*(.text)
```

Here the * is a wildcard which matches any file name. To exclude a list of files from matching the file name wildcard, EXCLUDE_FILE may be used to match all files except the ones specified in the EXCLUDE_FILE list. For example:

```
*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```

will cause all .ctors sections from all files except crtend.o and otherfile.o to be included.

There are two ways to include more than one section:

```
*(.text .rodata)
*(.text) *(.rodata)
```

The difference between these is the order in which the .text and .rodata input sections will appear in the output section. In the first example, they will be intermingled. In the second example, all .text input sections will appear first, followed by all .rodata input sections.

A file name can be specified to include sections from a particular file. This would be useful if one of the files contain special data that needs to be at a particular location in memory. For example:

```
data.o(.data)
```

If a file name is specified without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may be useful on occasion. For example:

```
data.o
```

When a file name is specified which does not contain any wild card characters, the linker will first see if the file name was also specified on the linker command line or in an INPUT command. If not, the linker will attempt to open the file as an input file, as though it appeared on the command line. This differs from an INPUT command because the linker will not search for the file in the archive search path.

8.6.5.2 INPUT SECTION WILDCARD PATTERNS

In an input section description, either the file name or the section name or both may be wildcard patterns.

The file name of * seen in many examples is a simple wildcard pattern for the file name.

The wildcard patterns are like those used by the UNIX shell.

*	matches any number of characters
?	matches any single character
[*chars*]	matches a single instance of any of the *chars*; the - character may be used to specify a range of characters, as in [a-z] to match any lower case letter
\	quotes the following character

When a file name is matched with a wildcard, the wildcard characters will not match a / character (used to separate directory names on UNIX). A pattern consisting of a single * character is an exception; it will always match any file name, whether it contains a / or not. In a section name, the wildcard characters will match a / character.

File name wildcard patterns only match files which are explicitly specified on the command line or in an INPUT command. The linker does not search directories to expand wild cards.

If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker will use the first match in the linker script. For example, this sequence of input section descriptions is probably in error, because the data.o rule will not be used:

```
.data  : { *(.data) }
.data1 : { data.o(.data) }
```

Normally, the linker will place files and sections matched by wild cards in the order in which they are seen during the link. This can be changed by using the SORT keyword, which appears before a wildcard pattern in parentheses (e.g., SORT(.text*)). When the SORT keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

To verify where the input sections are going, use the -M linker option to generate a map file. The map file shows precisely how input sections are mapped to output sections.

This example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all .text sections in .text and all .bss sections in .bss. The linker will place the .data section from all files beginning with an upper case character in .DATA; for all other files, the linker will place the .data section in .data.

```
   SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

8.6.5.3 INPUT SECTION FOR COMMON SYMBOLS

A special notation is needed for common symbols, because common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named COMMON.

File names may be used with the COMMON section just as with any other input sections. This will place common symbols from a particular input file in one section, while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the .bss section in the output file. For example:

```
   .bss { *(.bss) *(COMMON) }
```

If not otherwise specified, common symbols will be assigned to section .bss.

8.6.5.4 INPUT SECTION EXAMPLE

The following example is a complete linker script. It tells the linker to read all of the sections from file all.o and place them at the start of output section outputa which starts at location 0x10000. All of section .input1 from file foo.o follows immediately, in the same output section. All of section .input2 from foo.o goes into output section outputb, followed by section .input1 from foo1.o. All of the remaining .input1 and .input2 sections from any files are written to output section outputc.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

8.6.5.5 OUTPUT SECTION DESCRIPTION

The full description of an output section looks like this:

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

Most output sections do not use most of the optional section attributes.

The white space around *name* and *address* is required. The colon and the curly braces are also required. The line breaks and other white space are optional.

A section name may consist of any sequence of characters, but a name which contains any unusual characters such as commas must be quoted.

Each output-section-command may be one of the following:

- a symbol assignment (see **Section 8.6.3 “Assigning Values to Symbols”**)
- an input section description (see **Section 8.6.5.1 “Input Section Description”**)
- data values to include directly (see **Section 8.6.5.7 “Output Section Data”**)

8.6.5.6 OUTPUT SECTION ADDRESS

The *address* is an expression for the VMA (the virtual memory address) of the output section. If address is not provided, the linker will set it based on region if present, or otherwise based on the current value of the location counter.

If *address* is provided, the address of the output section will be set to precisely that. If neither *address* nor *region* is provided, then the address of the output section will be set to the current value of the location counter aligned to the alignment requirements of the output section. The alignment requirement of the output section is the strictest alignment of any input section contained within the output section.

For example,

```
.text . : { *(.text) }
```

and

```
.text : { *(.text) }
```

are subtly different. The first will set the address of the `.text` output section to the current value of the location counter. The second will set it to the current value of the location counter aligned to the strictest alignment of a `.text` input section.

The address may be an arbitrary expression (see **Section 8.7 “Expressions in Linker Scripts”**). For example, to align the section on a `0x10` byte boundary, so that the lowest four bits of the section address are zero, the command could look like this:

```
.text ALIGN(0x10) : { *(.text) }
```

This works because `ALIGN` returns the current location counter aligned upward to the specified value.

Specifying *address* for a section will change the value of the location counter.

8.6.5.7 OUTPUT SECTION DATA

Explicit bytes of data may be inserted into an output section by using BYTE, SHORT, LONG or QUAD as an output section command. Each keyword is followed by an expression in parentheses providing the value to store. The value of the expression is stored at the current value of the location counter.

The BYTE, SHORT, LONG and QUAD commands store one, two, four and eight bytes (respectively). For example, this command will store the four byte value of the symbol addr:

```
LONG(addr)
```

After storing the bytes, the location counter is incremented by the number of bytes stored. When using data commands in a program memory section, it is important to note that the linker considers program memory to be 32-bits wide, even though only 24 bits are physically implemented. Therefore, the most significant 8 bits of a LONG data value are not loaded into device memory.

Data commands only work inside a section description and not between them, so the following will produce an error from the linker:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

whereas this will work:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

The FILL command may be used to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the two least significant bytes of the expression, repeated as necessary. A FILL statement covers memory locations after the point at which it occurs in the section definition; by including more than one FILL statement, different fill patterns may be used in different parts of an output section.

This example shows how to fill unspecified regions of memory with the value 0x9090:

```
FILL(0x9090)
```

The FILL command is similar to the =fillexp output section attribute (see **Section 8.6.5.9 "Output Section Attributes"**), but it only affects the part of the section following the FILL command, rather than the entire section. If both are used, the FILL command takes precedence.

8.6.5.8 OUTPUT SECTION DISCARDING

The linker will not create an output section which does not have any contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. For example:

```
.foo { *(.foo) }
```

will only create a .foo section in the output file if there is a .foo section in at least one input file.

If anything other than an input section description is used as an output section command, such as a symbol assignment, then the output section will always be created, even if there are no matching input sections.

The special output section name /DISCARD/ may be used to discard input sections. Any input sections which are assigned to an output section named /DISCARD/ are not included in the output file.

8.6.5.9 OUTPUT SECTION ATTRIBUTES

To review, the full description of an output section is:

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

name, *address* and `output-section-command` have already been described. In the following sections, the remaining section attributes will be described.

8.6.5.10 OUTPUT SECTION TYPE

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

NOLOAD

The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.

DSECT, COPY, INFO, OVERLAY

These type names are supported for backward compatibility with older MIPS and GNU assemblers but are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.

The linker normally sets the attributes of an output section based on the input sections which map into it. This can be overridden by using the section type. For example, in the script sample below, the `ROM` section is addressed at memory location `0` and does not need to be loaded when the program is run. The contents of the `ROM` section will appear in the linker output file as usual.

```
SECTIONS {
  ROM 0 (NOLOAD) : { ... }
  ...
}
```

8.6.5.11 OUTPUT SECTION LMA

Every section has a virtual address (VMA) and a load address (LMA). The address expression which may appear in an output section description sets the VMA.

The linker will normally set the LMA equal to the VMA. This can be changed by using the `AT` keyword. The expression `lma` that follows the `AT` keyword specifies the load address of the section. Alternatively, with `AT>lma_region` expression, a memory region may be specified for the section's load address. See **Section 8.6.4 "MEMORY Command"**.

This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called `.text`, which starts at `0xBFC00000`, one called `.mdata`, which is loaded at the end of the `.text` section even though its VMA is `0xA0000000`, and one called `.bss` to hold uninitialized data at address `0xA0001000`. The symbol `_data` is defined with the value `0xA0000000`, which shows that the location counter holds the VMA value, not the LMA value.

```
SECTIONS
  {
  .text 0xBFC00000: { *(.text) _etext = . ; }
  .mdata 0xA0000000:
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0xA0001000:
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

The run-time initialization code for use with a program generated with this linker script would include a function to copy the initialized data from the ROM image to its run-time address. The initialization function could take advantage of the symbols defined by the linker script.

Writing such a function would rarely be necessary, however. These functions are provided by the C compiler's startup and initialization code. See the "*MPLAB® C/C++ Compiler User's Guide*" (DS51686) for more information on the startup code provided with the compiler. The assembly source code for the startup routine is provided in `\pic32-libs\c\startup\crt0.S`.

8.6.5.12 OUTPUT SECTION REGION

A section can be assigned to a previously defined region of memory by using `>region`. See **Section 8.6.4 "MEMORY Command"**.

Here is a simple example:

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

8.6.5.13 OUTPUT SECTION FILL

A fill pattern can be set for an entire section by using `=fillexp`. `fillexp` as an expression. Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the two least significant bytes of the value, repeated as necessary.

The fill value can also be changed with a `FILL` command in the output section commands; see **Section 8.6.5.7 "Output Section Data"**.

Here is a simple example:

```
SECTIONS { .text : { *(.text) } =0x9090 }
```

8.6.5.14 OVERLAY DESCRIPTION

An overlay description provides an easy way to describe sections which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the run-time memory address as required, perhaps by simply manipulating addressing bits.

Overlays are described using the OVERLAY command. The OVERLAY command is used within a SECTIONS command, like an output section description. The full syntax of the OVERLAY command is as follows:

```
 OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
{
  secname1
    {
      output-section-command
      output-section-command
      ...
    } [:phdr...] [=fill]
  secname2
    {
      output-section-command
      output-section-command
      ...
    } [:phdr...] [=fill]
  ...
} [>region] [:phdr...] [=fill]
```

Everything is optional except OVERLAY (a keyword), and each section must have a name (secname1 and secname2 above). The section definitions within the OVERLAY construct are identical to those within the general SECTIONS construct, except that no addresses and no memory regions may be defined for sections within an OVERLAY.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the OVERLAY as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to the current value of the location counter).

If the NOCROSSREFS keyword is used, and there are any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another.

For each section within the OVERLAY, the linker automatically defines two symbols. The symbol __load_start_secname is defined as the starting load address of the section. The symbol __load_stop_secname is defined as the final load address of the section. Any characters within secname which are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary.

At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section.

Here is an example. Remember that this would appear inside a SECTIONS construct.

```
OVERLAY 0x9D001000 : AT (0xA0004000)
 {
   .text0 { o1/*.o(.text) }
   .text1 { o2/*.o(.text) }
 }
```

This will define both .text0 and .text1 to start at address 0x9D001000. .text0 will be loaded at address 0x9D001000, and .text1 will be loaded immediately after .text0. The following symbols will be defined: __load_start_text0, __load_stop_text0, __load_start_text1, __load_stop_text1.

C code to copy overlay .text1 into the overlay area might look like the following:

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x9D001000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

The OVERLAY command is a convenience, since everything it does can be done using the more basic commands. The above example could have been written identically as follows.

```
.text0 0x9D001000: AT (0x9D004000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x9D001000: AT(0x9D004000+SIZEOF(.text0))
{o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x9D001000+ MAX (SIZEOF (.text0), SIZEOF (.text1));
```

8.6.6 Other Linker Script Commands

There are several other linker script commands, which are described briefly:

ASSERT(*exp, message*)

Ensure that *exp* is non-zero. If it is zero, then exit the linker with an error code, and print *message*.

ENTRY(*symbol*)

Specify *symbol* as the first instruction to execute in the program. The linker will record the address of this symbol in the output object file header. This does not affect the Reset instruction at address zero, which must be generated in some other way. By convention, the 32-bit linker scripts construct a GOTO __reset instruction at address zero.

EXTERN(*symbol symbol ...*)

Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. Several symbols may be listed for each EXTERN, and EXTERN may appear multiple times. This command has the same effect as the -u command line option.

FORCE_COMMON_ALLOCATION

This command has the same effect as the -d command line option: to make 32-bit linker assign space to common symbols even if a relocatable output file is specified (-r).

NOCROSSREFS(*section section ...*)

This command may be used to tell 32-bit linker to issue an error about any references among certain output sections. In certain types of programs, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors.

The NOCROSSREFS command takes a list of output section names. If the linker detects any cross references between the sections, it reports an error and returns a non-zero exit status. The NOCROSSREFS command uses output section names, not input section names.

OUTPUT_ARCH(*bfdarch*)

Specify a particular output machine architecture. The `bfdarch` value is always `pic32mx` for Microchip PIC32 MCUs.

OUTPUT_FORMAT(*format_name*)

The `OUTPUT_FORMAT` command names the object file format to use for the output file. The *`format_name`* value is always `elf32-tradlittlemips` for Microchip PIC32 MCUs.

TARGET(*format_name*)

The `TARGET` command names the object file format to use when reading input files. It affects subsequent `INPUT` and `GROUP` commands. The *`format_name`* value should remain `elf32-tradlittlemips` for Microchip PIC32 MCUs.

8.7 EXPRESSIONS IN LINKER SCRIPTS

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as 32-bit integers.

You can use and set symbol values in expressions.

The linker defines several special purpose built-in functions for use in expressions.

8.7.1 Constants

All constants are integers.

As in C, the linker considers an integer beginning with `0` to be octal, and an integer beginning with `0x` or `0X` to be hexadecimal. The linker considers other integers to be decimal.

In addition, you can use the suffixes `K` and `M` to scale a constant by 1024 or 1024*1024 respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

8.7.2 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword by surrounding the symbol name in double quotes:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, `A-B` is one symbol, whereas `A - B` is an expression involving subtraction.

8.7.3 The Location Counter

The special linker variable dot '.' always contains the current output location counter. Since the . always refers to a location in an output section, it may only appear in an expression within a SECTIONS command. The '.' symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to '.' will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x1234;
}
```

In the previous example, the .text section from file1 is located at the beginning of the output section output. It is followed by a 1000 byte gap. Then the .text section from file2 appears, also with a 1000 byte gap following before the .text section from file3. The notation = 0x1234 specifies what data to write in the gaps.

'.' actually refers to the byte offset from the start of the current containing object. Normally this is the SECTIONS statement, whose start address is 0, hence '.' can be used as an absolute address. If '.' is used inside a section description, however, it refers to the byte offset from the start of that section, not an absolute address. So, in a script like this:

```
SECTIONS
{
    . = 0x100
    .text: {
      *(.text)
      . = 0x200
    }
    . = 0x500
    .data: {
      *(.data)
      . += 0x600
    }
}
```

the .text section will be assigned a starting address of 0x100 and a size of exactly 0x200 bytes, even if there is not enough data in the .text input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move '.' backwards). The .data section will start at 0x500 and it will have an extra 0x600 bytes worth of space after the end of the values from the .data input sections and before the end of the .data output section itself.

8.7.4 Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels:

TABLE 8-2: PRECEDENCE OF OPERATORS

Precedence	Associativity	Operators	Description
1 (highest)	left	`! - ~`	Prefix operators
2	left	`* / %`	multiply, divide, modulo
3	left	`+ -`	add, subtract
4	left	`>> <<`	bit shift right, left
5	left	`== != > < <= >=`	Relational
6	left	`&`	bitwise and
7	left	`|`	bitwise or
8	left	`&&`	logical and
9	left	`||`	logical or
10	right	`? :`	Conditional
11 (lowest)	right	`&= += -= *= /=`	Symbol assignments

8.7.5 Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and lengths of memory regions, in order to do any linking at all. These values are computed as soon as possible when the linker reads in the linker script.

However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

The sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation.

Some expressions, such as those depending upon the location counter '.', must be evaluated during section allocation.

If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following:

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

will cause the error message "non-constant expression for initial address".

8.7.6 The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.

The position of the expression within the linker script determines whether it is absolute or relative. An expression which appears within an output section definition is relative to the base of the output section. An expression which appears elsewhere will be absolute.

A symbol set to a relative expression will be relocatable if you request relocatable output using the `-r` option. That means that a further link operation may change the value of the symbol. The symbol's section will be the section of the relative expression.

A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.

You can use the built-in function `ABSOLUTE` to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section `.data`:

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

If `ABSOLUTE` were not used, `_edata` would be relative to the `.data` section.

8.7.7 Built-in Functions

The linker script language includes a number of built-in functions for use in linker script expressions.

8.7.7.1 ABSOLUTE(*EXP*)

Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *exp*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative.

8.7.7.2 ADDR(*SECTION*)

Return the absolute address (the VMA) of the named section. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
...
}
```

8.7.7.3 ALIGN(*EXP*)

Return the location counter (.) aligned to the next exp boundary. *exp* must be an expression whose value is a power of two. This is equivalent to

```
(. + exp - 1) & ~(exp - 1)
```

ALIGN doesn't change the value of the location counter; it just does arithmetic on it. Here is an example which aligns the output .data section to the next 0x2000 byte boundary after the preceding section and sets a variable within the section to the next 0x8000 boundary after the input sections:

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
...
}
```

The first use of ALIGN in this example specifies the location of a section because it is used as the optional address attribute of a section definition (see **Section 8.6.5 "SECTIONS Command"**). The second use of ALIGN is used to define the value of a symbol.

The built-in function NEXT is closely related to ALIGN.

8.7.7.4 BLOCK(*EXP*)

This is a synonym for ALIGN, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

8.7.7.5 DEFINED(*SYMBOL*)

Return 1 if symbol is in the linker global symbol table and is defined; otherwise return 0. You can use this function to provide default values for symbols. For example, the following script fragment shows how to set a global symbol begin to the first location in the .text section, but if a symbol called begin already existed, its value is preserved:

```
SECTIONS { ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
  ...
}
```

8.7.7.6 KEEP(*SECTION*)

When link-time garbage collection is in use (--gc-sections), marking sections that should not be eliminated is often useful. This is accomplished by surrounding an input section's wildcard entry with KEEP(), as in KEEP(*(.init)) or KEEP(SORT_BY_NAME(*)(.ctors)).

8.7.7.7 LOADADDR(*SECTION*)

Return the absolute LMA of the named section. This is normally the same as ADDR, but it may be different if the AT attribute is used in the output section definition (see **Section 8.6.5 "SECTIONS Command"**).

8.7.7.8 MAX(*EXP1*, *EXP2*)

Returns the maximum of *exp1* and *exp2*.

8.7.7.9 MIN(*EXP1*, *EXP2*)

Returns the minimum of *exp1* and *exp2*.

8.7.7.10 NEXT(*EXP*)

Return the next unallocated address that is a multiple of *exp*. This function is equivalent to `ALIGN(exp)`.

8.7.7.11 SIZEOF(*SECTION*)

Return the size in bytes of the named section, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
...
}
```

NOTES:

Chapter 9. Linker Processing

9.1 INTRODUCTION

How the MPLAB XC32 Object Linker (xc32-ld) builds an application from input files is discussed here.

Topics covered in this chapter are:

- Overview of Linker Processing
- Linker Allocation
- Global and Weak Symbols
- Initialized Data
- Stack Allocation
- Heap Allocation
- PIC32MX Interrupt Vector Tables
- Interrupt Vector Tables for PIC32 MCUs Featuring Dedicated Programmable Variable Offsets

9.2 OVERVIEW OF LINKER PROCESSING

A linker combines one or more object files, with optional archive files, into a single executable output file. The object files contain relocatable sections of code and data which the linker will allocate into target memory. The entire process is controlled by a linker script, also known as a link command file. A linker script is required for every link.

The link process may be broken down into 5 steps:

1. Loading Input Files
2. Allocating Memory
3. Resolving Symbols
4. Computing Absolute Addresses
5. Building the Output File

9.2.1 Loading Input Files

The initial task of the linker is to interpret link command options and load input files. If a linker script is specified, that file is opened and interpreted. Otherwise an internal default linker script is used. In either case, the linker script provides a description of the target device, including specific memory region information. See **Chapter 8. "Linker Scripts"** for more details.

Next the linker opens all of the input object files. Each input file is checked to make sure the object format is compatible. If the object format is not compatible, an error is generated. The contents of each input file are then loaded into internal data structures. Typically each input file will contain multiple sections of code or data. Each section contains a list of relocation entries which associate locations in a section's raw data with relocatable symbols.

9.2.2 Allocating Memory

After all of the input files have been loaded, the linker allocates memory. This is accomplished by assigning each input section to an output section. The relation between input and output sections is defined by a section map in the linker script. An output section may or may not have the same name as an input section. Each output section is then assigned to a memory region in the target device.

> **Note:** Input sections are derived from source code by the compiler or the assembler. Output sections are created by the linker.

If an input section is not explicitly assigned to an output section, the linker will allocate the unassigned section according to section attributes. For more information about linker allocation, see **Section 9.3 "Linker Allocation"**.

9.2.3 Resolving Symbols

Once memory has been allocated, the linker begins the process of resolving symbols. Symbols defined in each input section have offsets that are relative to the beginning of the section. The linker converts these values into output section offsets.

Next, the linker attempts to match all external symbol references with a corresponding symbol definition. Multiple definitions of the same external symbol result in an error. If an external symbol is not found, an attempt is made to locate the symbol definition in an archive file. If the symbol definition is found in an archive, the corresponding archive module is loaded.

Modules loaded from archives may contain additional symbol references, so the process continues until all external symbol references have matching definitions. External symbols that are defined as "weak" receive special processing, as explained in **Section 9.4 "Global and Weak Symbols"**. If any external symbol reference

remains undefined, an error is generated.

9.2.4 Creating Special Sections

After the symbols have been resolved, the linker constructs any special input or output sections that are required. For example, the linker constructs a special input section named `.dinit` to support initialized data. Section.dinit is an initialization template that is interpreted by the C run-time library. For more information about initialized data, see **Section 9.5 “Initialized Data”**.

9.2.5 Computing Absolute Addresses

After the special sections have been created, the final sizes of all output sections are known. The linker then computes absolute addresses for all output sections and external symbols. Each output section is checked to make sure it falls within its assigned memory regions. If any section falls outside of its memory region, an error is generated. Any symbols defined in the linker script are also computed.

9.2.6 Building the Output File

Finally, the linker builds the output file. Relocation entries in each section are patched using absolute addresses. If the address computed for a symbol does not fit in the relocation entry, a link error results. This can occur, for example, when one module references a variable which it thinks is in a “small data” section, while the other defines it in a non-small section.

A link map is also generated if requested with the appropriate option. The link map includes a memory usage report, which shows the starting address and length of all sections in data memory and program memory. For more information about the link map, see **Section 6.4.5 “Map File”**.

9.3 LINKER ALLOCATION

Linker allocation is controlled by the linker script, and proceeds in three steps:

1. Mapping Input Sections to Output Sections
2. Assigning Output Sections to Regions
3. Allocating Unmapped Sections

Steps 1 and 2 are performed by a sequential memory allocator. Input sections which appear in the linker script are assigned to specific memory regions in the target devices. Addresses within a memory region are allocated sequentially, beginning with the lowest address and growing upwards.

Step 3 is performed by a best-fit memory allocator. Input sections which do not appear in the linker script are assigned to memory regions according to their attributes. The best-fit allocator makes efficient use of any remaining memory, including gaps between output sections that may have been left by the sequential allocator.

9.3.1 Mapping Input Sections to Output Sections

Input sections are grouped and mapped into output sections, according to the section map. When an output section contains several different input sections, the exact ordering of input sections may be important. For example, consider the following output section definition:

```
/* Code Sections */
.text ORIGIN(kseg0_program_mem)  :
{
  *(.text .stub .text.* .gnu.linkonce.t.*)
  *(.mips16.fn.*)
  *(.mips16.call.*)
} >kseg0_program_mem =0
```

Here the output section named `.text` is defined. Notice that the contents of this section are specified within curly braces {}. After the closing brace, `>kseg0_program_mem` indicates that this output section should be assigned to memory region `kseg0_program_mem`.

The contents of output section `.text` may be interpreted as follows:

- Input sections named `.text` and `.stub` and input sections that match the wild-card patterns `.text.*` and `.gnu.linkonce.t.*` are collected and mapped into the output section. Grouping these sections ensures locality of reference.
- Input sections that match the wildcard pattern `.mips16.fn.*` are collected and mapped into the output section.
- Input sections that match the wildcard pattern `.mips16.call.*` are collected and mapped into the output section.

9.3.2 Assigning Output Sections to Regions

Once the sizes of all output sections are known, they are assigned to memory regions. Normally a region is specified in the output section definition. If a region is not specified, the first defined memory region will be used.

Memory regions are filled sequentially, from lower to higher addresses, in the same order that sections appear in the section map. A location counter, unique to each region, keeps track of the next available memory location. There are two conditions which may cause gaps in the allocation of memory within a region:

1. The section map specifies an absolute address for an output section, or
2. The output section has a particular alignment requirement.

In either case, any intervening memory between the current location counter and the absolute (or aligned) address is skipped. The exact address of all items allocated in memory may be determined from the link map file.

For a section containing an aligned memory block (with the `aligned` attribute in C or `.align` directive in assembly), the section must also be aligned, to the same (or greater) alignment value. If two or more input sections have different alignment requirements, the largest alignment is used for the output section.

9.3.3 Allocating Unmapped Sections

After all sections that appear in the section map are allocated, any remaining sections are considered to be unmapped. Unmapped sections are allocated according to section attributes. The linker uses a best-fit memory allocator to determine the most efficient arrangement in memory. The primary emphasis of the best-fit allocator is the reduction or elimination of memory gaps due to address alignment restrictions. By convention, most standard sections such as the `.text`, `.data`, `.bss`, or `.ramfunc` section are not explicitly mapped in linker scripts, thus providing maximum flexibility for the best-fit memory allocator. The exception is the “small” data sections used for gp-relative addressing. Because these sections must be grouped together, they are mapped in the linker script. A future toolchain release may allow the “small” data sections to be allocated by the best-fit allocator.

Section attributes affect memory allocation as described below. For a general discussion of section attributes, see **Section A.2 “Assembler Directives that Define Sections”**.

code

The code attribute specifies that a section should be allocated in program memory, as defined by region kseg0_program_mem in the linker script. The following attributes may be used in conjunction with code and will further specify the allocation:

- address() specifies an absolute address
- align() specifies alignment of the section starting address

data

The data attribute specifies that a section should be allocated as initialized storage in data memory, as defined by regions kseg0_data_mem & kseg1_data_mem in the linker script. The following attributes may be used in conjunction with data and will further specify the allocation:

- address() specifies an absolute address
- near specifies the first 64K of data memory
- align() specifies alignment of the section starting address
- reverse() specifies alignment of the section ending address + 1

bss

The bss attribute specifies that a section should be allocated as uninitialized storage in data memory, as defined by region kseg0_data_mem & kseg1_data_mem in the linker script. The following attributes may be used in conjunction with bss and will further specify the allocation:

- address() specifies an absolute address
- near specifies the first 64K of data memory
- align() specifies alignment of the section starting address
- reverse() specifies alignment of the section ending address + 1

persist

The persist attribute specifies that a section should be allocated as persistent storage in data memory, as defined by region kseg0_data_mem & kseg1_data_mem in the linker script. Persistent storage is not cleared or initialized by the C run-time library. The following attributes may be used in

- conjunction with persist and will further specify the allocation:
- address() specifies an absolute address
- near specifies the first 64K of data memory
- align() specifies alignment of the section starting address
- reverse() specifies alignment of the section ending address + 1

9.4 GLOBAL AND WEAK SYMBOLS

When a symbol reference appears in an object file without a corresponding definition, the symbol is declared external. By default, external symbols have global binding and are referred to as global symbols. External symbols may be explicitly declared with weak binding, using the `__weak__` attribute in C or the `.weak` directive in assembly language.

As the name implies, global symbols are visible to all input files involved in the link. There must be one (and only one) definition for every global symbol referenced. If a global definition is not found among the input files, archives will be searched and the first archive module found that contains the needed definition will be loaded. If a definition is not found for a global symbol, a link error is reported.

Weak symbols share the same name space as global symbols, but are handled differently. Multiple definitions of a weak symbol are permitted. If a weak definition is not found among the input files, archives are not searched and a value of 0 is assumed for all references to the weak symbol. A global symbol definition of the same name will take precedence over a weak definition (or the lack of one). In essence, weak symbols are considered optional and may be replaced by global symbols, or ignored entirely.

9.5 INITIALIZED DATA

The linker provides automatic support for initialized variables in data memory. Variables are allocated in sections. Each data section is declared with a flag that indicates whether it is initialized, or not initialized.

To control the initialization of the various data sections, the linker constructs a data initialization template. The template is allocated in program memory, and is processed at start-up by the run-time library. When the application main program takes control, all variables in data memory have been initialized.

- Standard Data Section Names
- Data Initialization Template
- Run-Time Library Support

9.5.1 Standard Data Section Names

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

TABLE 9-1: TRADITIONAL SECTION NAMES

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

The name "bss" dates back several decades, and means memory "Block Started by Symbol". By convention, bss memory is filled with zeros during program start-up. The traditional section names are considered to have implied attributes as listed in Table 9-1. The code attribute indicates that the section contains executable code and should be loaded in program memory. The bss attribute indicates that the section contains data storage that is not initialized, but will be filled with zeros at program start-up. The data attribute indicates that the section contains data storage that receives initial values at start-up.

Assembly applications may define additional sections with explicit attributes using the section directive described in Section "Directives that Define Sections". For C applications, the 32-bit compiler will automatically define sections to contain variables and functions as needed. For more information on the attributes of variables and functions that may result in automatic section definition, see the "*MPLAB XC32 C/C++ Compiler User's Guide"* (DS51686).

Note: Whenever a section directive is used, all declarations that follow are assembled into the named section. This continues until another section directive appears, or the end of file. For more information on defining sections and section attributes, see **Section X.Y "Directives that Define Sections"**.

9.5.2 Data Initialization Template

As noted in **Section 9.5.1 "Standard Data Section Names"**, the 32-bit Language Tools support BSS-type sections (memory that is not initialized) as well as data-type sections (memory that receives initial values). The data-type sections receive initial values at start-up, and the BSS-type sections are filled with zeros. A generic data initialization template is used that supports any number of arbitrary BSS-type sections or data-type sections. The data initialization template is created by the linker and is loaded into an output section named `.dinit` in program memory. Start-up code in the run-time library interprets the template and initializes data memory accordingly.

The data initialization template contains one record for each output section in data memory. The template is terminated by a null instruction word. The format of a data initialization record is:

```
/* data init record */
struct data_record {
char *dst; /* destination address */
unsigned int len; /* length in bytes */
unsigned int format:7; /* format code */
char dat[0]; /* variable length data */
};
```

The first element of the record is a pointer to the section in data memory. The second and third elements are the section length and format code, respectively. The last element is an optional array of data bytes. For BSS-type sections, no data bytes are required.

The format code has two possible values.

TABLE 9-2: FORMAT CODE VALUES

Format Code	Description
0	Fill the output section with zeros
1	Copy 4 bytes of data from each instruction word in the data array

9.5.3 Run-Time Library Support

In order to initialize variables in data memory, the data initialization template must be processed at start-up, before the application's main function takes control. For C programs, this task is performed by C start-up modules in the runtime library. Assembly language programs can also use the C start-up modules by linking with libpic32.a.

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

TABLE 9-3: TABLE MAIN ENTRY POINTS

Main Entry Name	Description
`_reset`	Takes control immediately after device Reset
`main`	Takes control after the start-up module completes its work

Note that the first entry name (_reset) includes one leading underscore characters. The second entry name (main) includes no leading underscore character. On device Reset, the startup module is called and it performs the following:

1. Initialize Stack Pointer
2. The data initialization template in section `.dinit` is read, causing all uninitialized sections to be cleared, and all initialized sections to be initialized with values read from program memory.
3. Copy RAM functions from program flash to data memory an initialize bus matrix registers.
4. The function main is called with no parameters.
5. If main returns, the processor will reset.

The alternate start-up module is linked when the --no-data-init option is specified.

It performs the same operations, except for step (2), which is omitted. The alternate start-up module is smaller than the primary module, and can be selected to conserve program memory if data initialization is not required.

Source code for both modules is provided in the src directory of the MPLAB XC32 C compiler installation directory. The start-up modules may be modified if necessary. For example, if an application requires main to be called with parameters, a conditional assembly directive may be switched to provide this support.

9.6 STACK ALLOCATION

The MPLAB C compiler for PIC32 MCUs dedicates general-purpose register 29 as the software Stack Pointer. All processor stack operations, including function calls, interrupts, and exceptions use the software stack. The stack grows downward from high addresses to low addresses.

By default, 32-bit linker dynamically allocates the largest stack possible from unused data memory. Previous releases used output sections specified in the linker script to allocate the stack.

The location and size of the stack is reported in the link map output file and the Memory-Usage Report, under the heading Dynamic Memory Usage. Applications can ensure that at least a minimum sized stack is available by specifying the size on the linker command line using the --defsym=_min_stack_size=size linker command line option. An example of allocating a stack of 2048 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym=_min_stack_size=2048.
The linker script a default _min_stack_size of 1024.
```

> **Note:** See the "*MPLAB XC32 C/C++ Compiler User's Guide"* (DS51686) for more information on the compiler's usage of the stack.

The linker's reported size of the `.stack` section is the minimum size required to avoid a link error. The effective stack size is usually larger than the reported `.stack` section size.

9.7 HEAP ALLOCATION

The C runtime heap is an uninitialized area of data memory that is used for dynamic memory allocation using the standard C library dynamic memory-management functions, calloc, malloc, and realloc. If you do not use any of these functions (directly or indirectly), then you do not need to allocate a heap. **By default, the heap size is 0**.

If you do want to use dynamic memory allocation, either directly, by calling one of the memory allocation functions, or indirectly, by using a standard C library function that uses one of these functions, then a heap must be created. A heap is created by specifying its size on the linker command line using the `--defsym=_min_heap_size` linker command line option. An example of allocating a heap of 512 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym=_min_heap_size=512
```

The linker allocates the heap immediately before the stack. The location and size of the heap are reported in the link map output file and in the Memory-Usage Report, under the heading Dynamic Memory Usage. If the requested size is not available, the linker reports an error.

The heap is now dynamically allocated by the linker. Previous releases used output sections specified in the linker script to allocate the heap.

9.8 PIC32MX INTERRUPT VECTOR TABLES

The vector address of a given interrupt is calculated using Exception Base (EBASE <31:12>) register, which provides a 4 KB page-aligned base address value located in the kernel segment (kseg) address space. (EBASE is a CPU register.) The address is calculated by using EBASE and VS (INTCTL <9:5>) values. The VS bits provide the vector spacing between adjacent vector addresses.

The linker script creates the corresponding Interrupt Vector Table as follows:

```
PROVIDE(_vector_spacing = 0x00000001);
_ebase_address  = 0x9FC01000;

SECTIONS
{
  .app_excpt _GEN_EXCPT_ADDR :
  {
    KEEP(*(.gen_handler))
  } > exception_mem
  .vector_0 _ebase_address + 0x200 :
  {
    KEEP(*(.vector_0))
  } > exception_mem
  ASSERT (_vector_spacing == 0 || SIZEOF(.vector_0) <=
     (_vector_spacing << 5),
     "function at exception vector 0 too large")
  .vector_1 _ebase_address + 0x200 +
     (_vector_spacing << 5) * 1 :
  {
    KEEP(*(.vector_1))
  } > exception_mem
  ASSERT (_vector_spacing == 0 || SIZEOF(.vector_1) <=
     (_vector_spacing << 5),
     "function at exception vector 1 too large")
  .vector_2 _ebase_address + 0x200 +
     (_vector_spacing << 5) * 2 :
  {
    KEEP(*(.vector_2))
  } > exception_mem
/* … */
  .vector_63 _ebase_address + 0x200 +
     (_vector_spacing << 5) * 63 :
  {
    KEEP(*(.vector_63))
  } > exception_mem
  ASSERT (_vector_spacing == 0 || SIZEOF(.vector_63) <=
     (_vector_spacing << 5),
     "function at exception vector 63 too large")
}
```

Each vector in the table is created as an output section located at an absolute address based on values of the `_ebase_address` and `_vector_spacing` symbols. There is one output section for each of the 64 vectors in the table.

9.9 INTERRUPT VECTOR TABLES FOR PIC32 MCUS FEATURING DEDICATED PROGRAMMABLE VARIABLE OFFSETS

Some PIC32 families feature variable offsets for vector spacing. This feature allows the interrupt vector spacing to be configured according to application needs. A specific interrupt vector offset can be set for each vector using its associated `OFFxxx` register. For details on the interrupt vector-table variable offset feature, refer to the PIC32 Family Reference Manual (DS61108) and also the data sheet for your specific PIC32 MCU.

The XC32 toolchain provides a device-specific default linker script and a corresponding object file that work together with the default runtime startup code. The following table shows the files, located in `/pic32mx/lib/proc/<devicename>`, that are used to initialized vector-table offset registers.

Device linker script	`<devicename>.ld`
Vector Offset initialization	`vector_offset_init.o`
Device runtime startup code	`crt0_<boot_isa>.o`

9.9.1 Device-specific Linker Script

The application source code is responsible for creating a `.vector_n` input section for each interrupt vector. The C/C++ compiler creates this section when either the `vector(n)` or `at_vector(n)` attribute is applied to the interrupt service routine. In assembly code, use the `.section` directive to create a new named section.

The device-specific linker script creates a single output section named `.vectors` that groups all of the individual `.vector_n` input sections from the project. The start of the interrupt-vector table is mapped to the address (`_ebase_address + 0x200`). The default value of the `_ebase_address` symbol is also provided in the linker script.

For each vector, the linker script also creates a symbol named `__vector_offset_n` whose value is the offset of the vector address from the `_ebase_address` address.

```
PROVIDE(_ebase_address = 0x9D000000);

SECTIONS
{
 /* Interrupt vector table with vector offsets */
 .vectors _ebase_address + 0x200 :
 {
  /*  Symbol __vector_offset_n points to .vector_n if it exists,
   *  otherwise points to the default handler. The
   *  vector_offset_init.o module then provides a .data section
   *  containing values used to initialize the vector-offset SFRs
   *  in the crt0 startup code.
   */
 __vector_offset_0 = (DEFINED(__vector_dispatch_0) ? (. - _ebase_address) : __vector_offset_default);
 KEEP(*(.vector_0))
 __vector_offset_1 = (DEFINED(__vector_dispatch_1) ? (. - _ebase_address) : __vector_offset_default);
 KEEP(*(.vector_1))
 __vector_offset_2 = (DEFINED(__vector_dispatch_2) ? (. - _ebase_address) : __vector_offset_default);
 KEEP(*(.vector_2))

 /* … */

 __vector_offset_190 = (DEFINED(__vector_dispatch_190) ? (. - _ebase_address) : __vector_offset_default);
 KEEP(*(.vector_190))
 }
}
```

9.9.2 Vector-Offset Initialization Module

The vector-offset initialization module (`vector_offset_init.o`) uses the `__vector_offset_n` symbols defined in the default linker script. The value of each symbol is the offset of the vector's address from the ebase register's address. The vector-offset initialization module, uses the symbol value to create a `.data` section using the address of the corresponding `OFFxxx` special function register. This means that the standard linker-generated data-initialization template contains the values used to initialize the `OFFxxx` registers.

```
  .section
.data.__vector_offset_BF810540,data,keep,address(0xBF810540)
  .word __vector_offset_0
  .word __vector_offset_1
  .word __vector_offset_2
  .word __vector_offset_3
  .word __vector_offset_4
  .word __vector_offset_5
  .word __vector_offset_6
```

9.9.3 Runtime Startup-Code Data Initialization

With these `.data` sections added to the project and the linker-generated data-initialization template, the standard runtime startup code initializes the `OFFxxx` special function registers as regular initialized data. No special code is required in the startup code to initialize the `OFFxxx` registers.

9.9.4 Example Vector-Table Assembly Source Code

The following example shows how to create a vector dispatch for interrupt vector 0. The vector dispatch is a jump from the vector table to the actual Interrupt Service Routine (ISR).

```
/*  Input section .vector_0 is mapped to the .vectors output
       *  section in the linker script.
       */
   .globl _vector_dispatch_0
   .section .vector_0,code
   .align 2
   .set nomips16
   .ent __vector_dispatch_0
__vector_dispatch_0:
      /* Jump to the actual ISR code */
   j  isrvector0
   nop
   .end __vector_dispatch_0
   .size __vector_dispatch_0, .-__vector_dispatch_0
```

The following example shows how to place the Interrupt Service Routine directly into the vector table. The mapping in the linker script will automatically adjust the vector spacing to accommodate the function's code.

```
/*  Input section .vector_0 is mapped to the .vectors output
       *  section in the linker script.
       */
      .section .vector_0,code
   .align 2
   .globl isrvector0
   .set nomips16
   .set nomicromips
   .ent isrvector0
isrvector0:
   .set noat

      /*  Interrupt Service Routine code directly in the vector table.
       *  Be sure to preserve registers as appropriate for an ISR.
       */

   eret
   .set at
   .end isrvector0
   .size isrvector0, .-isrvector0
```

From XC32 C code, use the standard `vector(n)` and `at_vector(n)` function attributes on your ISR function. Refer to the "*MPLAB XC32 C/C++ Compiler User's Guide*" (DS51686) for further information on these function attributes.

NOTES:

Chapter 10. Linker Examples

10.1 INTRODUCTION

The 32-bit compiler and assembler each provide a syntax that can be used to designate certain elements of an application for special handling. In C, a rich set of attributes are available to modify variable and function definitions (see the "*MPLAB XC32 C/C++ Compiler User's Guide"* (DS51686). In assembly language, variables and functions are abstracted into memory sections, which become inputs to the linker. The assembler provides another set of attributes that are available to modify section definitions (see **Section 4.6 "Directives that Modify Section Alignment"**).

This chapter includes a number of 32-bit specific linker examples and shows the equivalent syntax in C and assembly language.

10.2 HIGHLIGHTS

Topics covered in this chapter are:

- Memory Addresses and Relocatable Code
- Locating a Variable at a Specific Address
- Locating a Function at a Specific Address
- Locating and Reserving Program Memory

10.3 MEMORY ADDRESSES AND RELOCATABLE CODE

For most applications it is preferable to write fully relocatable source code, thus allowing the linker to determine the exact addresses in memory where functions and variables are placed. The final address of external symbols in memory can be determined from the link map output, as shown in this excerpt:

```
...
.text                          0x9d0000f0    0x64
 .text                         0x9d0000f0    0x64 test.o
                               0x9d0000f0    myfunc
                               0x9d000110    main

.text._DefaultInterrupt        0x9d000154    0x48
 .text._DefaultInterrupt       0x9d000154    0x48 c:/program files/
                                             microchip/xc32/v1.20/bin/
                                             ../lib/gcc/pic32mx/4.5.2/
                                             ../../../../pic32mx/
                                             lib\libpic32.a
                                             defaultinterrupt.o)
                               0x9d000154    _DefaultInterrupt...
```

In some cases it is necessary for the programmer to specify the address where a certain variable or function should be located. Traditionally this is done by creating a user-defined section and writing a custom linker script. The 32-bit assembler and compiler provide a set of attributes that can be used to specify absolute addresses and memory spaces directly in source code. When these attributes are used, custom linker scripts are not required.

Note: By specifying an absolute address, the programmer assumes the responsibility to ensure the specified address is reasonable and available. If the specified address is out of range, or conflicts with a statically allocated resource, a link error will occur.

10.4 LOCATING A VARIABLE AT A SPECIFIC ADDRESS

In this example, array `buf1` is located at a specific address in data memory. The address of `buf1` can be confirmed by executing the program in the simulator, or by examining the link map.

```
#include "stdio.h"
int __attribute__((address(0xa0000200))) buf1[128];
```

The equivalent array definition in assembly language appears below. The `.align` directive is optional and represents the default alignment in data memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
.globl                  buf1
                        .section    *,address(0xa0000200),bss
                        .align      2
                        .type       buf1, @object
                        .size       buf1, 512
buf1:
                        .space      512
```

10.5LOCATING A FUNCTION AT A SPECIFIC ADDRESS

In this example, function `func` is located at a specific address.

```
#include "stdio.h"
void __attribute__((address(0x9d002000))) func()
{}
```

The equivalent function definition in assembly language appears below. The `.align` directive is optional and represents the default alignment in program memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
.section                *,address(0x9d002000),code
                        .align      2
                        .globl      func
func:
                        ......
```

10.6 LOCATING AND RESERVING PROGRAM MEMORY

In this example, a block of program memory is reserved for a special purpose, such as a bootloader. An arbitrary sized function is allocated in the block, with the remaining space reserved for expansion or other purposes.

The following output section definition is added to a custom linker script:

```
BOOT_START = 0x9d00A200;
BOOT_LEN = 0x400;
my_boot BOOT_START :
{
*(my_boot);
. = BOOT_LEN; /* advance dot to the maximum length */
} > kseg0_program_mem
```

Note the "dot assignment" (.=) that appears inside the section definition after the input sections. Dot is a special variable that represents the location counter, or next fill point, in the current section. It is an offset relative to the start of the section. The statement in effect says "no matter how big the input sections are, make sure the output section is full size."

The following C function will be allocated in the reserved block:

```
void __attribute__((section("my_boot"))) func1()
{
  /* etc. */
}
```

The equivalent assembly language would be:

```
 .section        my_boot,code
        .align  2
        .globl  func1
func1:
                              ......
```

Chapter 11. Linker Errors/Warnings

11.1 INTRODUCTION

MPLAB XC32 Object Linker (xc32-ld) generates errors and warnings. A descriptive list of these outputs is shown here. This list shows only the most common diagnostic messages from the linker.

Topics covered in this appendix are:

- Fatal Errors
- Errors
- Warnings

11.2 FATAL ERRORS

The following errors indicate that an internal error has occurred in the linker. If the linker emits any of the fatal errors listed below and you're using a custom linker script, check that the script specifies `OUTPUT_FORMAT` (`elf32-tradlittlemips`) and `OUTPUT_ARCH` (`pic32mx`). Other values may cause the linker to operate in an unsupported mode. Also check that you are passing only fully supported options on the linker's command line. Finally, make sure that no other applications have the linker's input or output files locked.

If the fatal error occurs with the correct `OUTPUT_FORMAT`, `OUTPUT_ARCH`, and command-line options, contact Microchip Technology at http://support.microchip.com for engineering support. Be sure to provide full details about the source code and command-line options that caused the error.

- Bfd backend error: bfd_reloc_ctor unsupported
- Bfd_hash_allocate failed creating symbol %s
- Bfd_hash_lookup failed: %e
- Bfd_hash_lookup for insertion failed: %e
- Bfd_hash_table_init failed: %e
- Bfd_hash_table_init of cref table failed: %e
- Bfd_link_hash_lookup failed: %e
- Bfd_new_link_order failed
- Bfd_record_phdr failed: %e
- Can't set bfd default target to `%s': %e
- Can not create link hash table: %e
- Can not make object file: %e
- Cannot represent machine `%s'
- Could not read relocs: %e
- Could not read symbols
- Cref_hash_lookup failed: %e
- Error closing file `%s'
- Error writing file `%s'
- Failed to create hash table
- Failed to merge target specific data
- File not recognized: %e
- Final close failed: %e
- Final link failed: %e
- Hash creation failed
- Out of memory during initialization
- Symbol `%t' missing from main hash table
- Target %s not found
- Target architecture respecified
- Unknown architecture: %s
- Unknown demangling style `%s'
- Unknown language `%s' in version information

11.3 ERRORS

The linker errors listed below usually indicate an error in the linker script or command-line options passed to the linker. An error could also indicate a problem with one or more of the input object files or archives.

Symbols

--gc-sections and -r may not be used together

The garbage-collection sections option and the relocatable-output option are not compatible. Remove either the `--gc-sections` option or the `--relocatable` option.

--relax and -r may not be used together

The relaxation option and the relocatable output option are not compatible. Remove one of the options.

A

A heap is required, but has not been specified.

A heap must be specified when using Standard C input/output functions

Assignment to location counter invalid outside of SECTION

An assignment to the special dot symbol can be done only during allocation within a `SECTION`. Check the location of the assignment statement in the linker script.

B

Bad --unresolved-symbols option: *option*

The `--unresolved-symbols` method option was invalid. Note that this option is unsupported. Try the default `--unresolved-symbols=report-all` instead.

C

Can not PROVIDE assignment to location counter

An assignment to the special dot symbol can be done only during allocation. A *PROVIDE* command cannot use an assignment to the location counter. Remove the erroneous statement from the linker script.

Can not set architecture: *arch_name*

If you're using a custom linker script, check that the `OUTPUT_ARCH` (`pic32mx`) command appears in the linker script. The PIC32 MCU linker currently supports only the 'pic32mx' arch.

Cannot move location counter backwards (from *addr1* to *addr2*)

The next dot-symbol value must be greater than the current dot-symbol value.

Could not allocate data memory.

The linker could not find a way to allocate all of the sections that have been assigned to region ‘kseg0_data_memory/kseg1_data_memory’.

Could not allocate program memory.

The linker could not find a way to allocate all of the sections that have been assigned to region ‘kseg0_program_memory’.

D

Dangerous relocation: *relocation_type*

A symbol was resolved but the usage is dangerous. This can occur, for example, when the code uses GP-relative addressing but the `_gp` initialization symbol was not defined. The `_gp` symbol is normally defined in the linker script.

--data-init and --no-data-init options can not be used together.

`--data-init` creates a special output section named `.dinit` as a template for the run-time initialization of data, `--no-data-init` does not. Only one option can be used.

F

File format not recognized; treating as linker script

One of the input files was not a recognized ELF object or archive. The linker assumes that it is a linker script.

G

Group ended before it began (--help for usage)

The `-)` option appeared on the command line before the `-(` option. Check that the group is specified correctly on the linker command line.

I

Illegal use of *name* section

The section name is reserved. For instance, the special output-section name `/DISCARD/` may be used to discard input sections. Any input sections which are assigned to an output section named `/DISCARD/` are not included in the output file. You should not create your own output section named `/DISCARD/`.

Includes nested too deeply

The maximum include depth is 10.

Invalid argument to option --section-start

The argument to `--section-start` must be `sectionname=`*`org`*. *`org`* must be a single hexadecimal integer. There should be no white space between `sectionname`, the equals sign (`=`), and *`org`*.

Invalid assignment to location counter

The assignment to the special dot symbol was invalid.

Invalid syntax in flags

The section flags are invalid. The accepted flags are: `a r w x l`.

M

Macros nested too deeply

The maximum macro nesting depth is 10.

May not nest groups (--help for usage)

An archive group is already started. Use the `-)` option to close the current group before starting another group with the `-(` option.

Member %b in archive is not an object

The archive member is not a valid object. Check that the library archive is correct for the Microchip MPLAB XC32 C/C++ Compiler.

Missing argument(s) to option --section-start

The required argument to `--section-start` must be `sectionname=`*`org`*.

Multiple definition of *name*

The linker discovered a symbol that is defined multiple times. Eliminate the extraneous definition(s).

Multiple startup files

The linker script is attempting to set a startup file, but a startup file has already been set. There should be only one startup file specified in the linker script.

N

No input files

The linker did not find an input file specified on the command line. There was nothing for the linker to do. Check that you are passing the correct object file names to the linker.

Nonconstant expression for *name*

`name` must be a nonconstant expression.

Not enough memory for stack (num bytes available).

There was not enough memory free to allocate the minimum-sized stack.

R

region *region* is full (*filename* section *secname*).

The memory region *`region`* is full, but section *`secname`* has been assigned to it.

Reloc refers to symbol *name* which is not being output

An instruction references a symbol that is not being output.

Relocation truncated to fit *relocation_type name*.

This error indicates that the relocated value of name is too large for its intended use. This can happen when an address is out of range for the instruction in question. Check that the symbol is both declared and defined in the intended section. For instance, a variable's declaration and definition must both be either const or non-const.

Relocation truncated to fit: *relocation_type name* against undefined symbol *name*

This error can occur if the symbol does not exist. For instance, the code calls a function that has not been defined.

U

Undefined MEMORY region *region* referenced in expression

The expression referenced a `MEMORY` region that does not exist in the linker script.

11.4 WARNINGS

The linker generates warnings when an assumption is made so that the linker could continue linking a flawed program. Warnings should not be ignored. Each warning should be specifically looked at and corrected to ensure that the linker understands what was intended. Warning messages can sometimes point out bugs in your program.

C

Cannot find entry symbol *name*

If the linker cannot find the specified entry symbol and it is not a number. Use the first address in the text section.

Changing start of section *name* by *num* bytes

The linker is changing the start of the indicated section due to alignment.

D

data initialization has been turned off, therefore section secname will not be initialized.

The specified section requires initialization, but data initialization has been turned off; so, the initial data values are discarded. Storage for the data sections will be allocated as usual.

I

initial values were specified for a non-loadable data section (name). These values will be ignored.

By definition, a persistent data section implies data that is not initialized; therefore the values are discarded. Storage for the section will be allocated as usual.

R

Redeclaration of memory region *name*

The `MEMORY` region has been declared more than once in the linker script.

U

Undefined reference to *name*

The symbol is undefined.

Part 3 – 32-Bit Utilities (including the Archiver/Librarian)

NOTES:

Chapter 12. MPLAB XC32 Object Archiver/Librarian

12.1 INTRODUCTION

The MPLAB XC32 Object Archiver/Librarian (xc32-ar) creates, modifies and extracts files from archives. This tool is one of several utilities. An "archive" is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called "members" of the archive).

The 32-bit archiver/librarian can maintain archives whose members have names of any length; however, if an `f` modifier is used, the file names will be truncated to 15 characters.

The archiver is considered a binary utility because archives of this sort are most often used as "libraries" holding commonly needed subroutines.

The archiver creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier `s`. Once created, this index is updated in the archive whenever the archiver makes a change to its contents (save for the `q` update operation). An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

You may use `xc32-nm -s` or `xc32-nm --print-armap` to list this index table. If an archive lacks the table, another form of the 32-bit archiver/librarian called `xc32-ranlib` can be used to add only the table.

The 32-bit archiver/librarian is designed to be compatible with two different facilities. You can control its activity using command-line options or, if you specify the single command line option `-M`, you can control it with a script supplied via standard input.

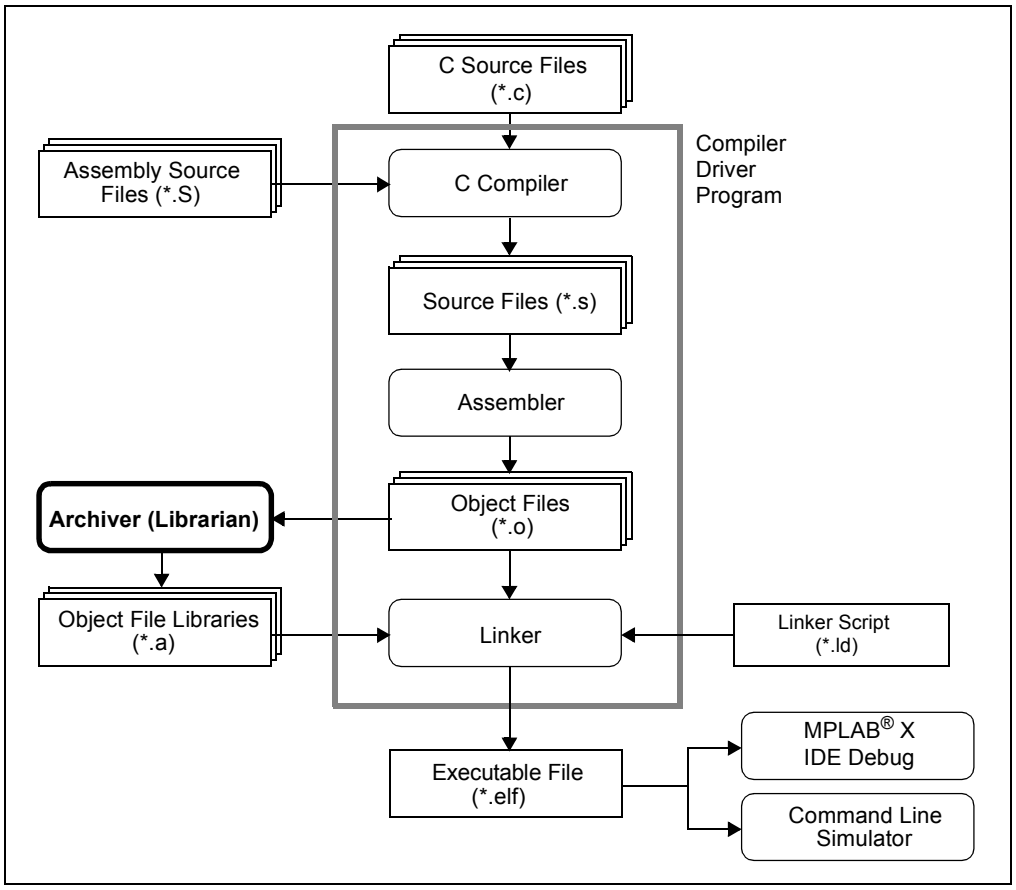
Topics covered in this chapter are:

- Archiver/Librarian and Other Development Tools
- Feature Set
- Input/Output Files
- Syntax
- Options
- Scripts

12.2 ARCHIVER/LIBRARIAN AND OTHER DEVELOPMENT TOOLS

The 32-bit librarian creates an archive file from object files created by the 32-bit assembler. Archive files may then be linked by the 32-bit linker with other relocatable object files to create an executable file. See Figure 12-1 for an overview of the tools process flow.

FIGURE 12-1: MPLAB X IDE **TOOLS PROCESS FLOW**



12.3 FEATURE SET

Notable features of the assembler include:

- Available for Linux x86, Mac OS X, and Windows
- Command Line Interface

12.4 INPUT/OUTPUT FILES

The 32-bit archiver/librarian generates archive files (`.a`). An archive file is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive). All objects are processed in the ELF object-file format.

xc32-ar is considered a binary utility because archives of this sort are most often used as libraries holding commonly needed subroutines.

12.5 SYNTAX

```
xc32-ar [-]P[MOD [RELPOS] [COUNT]] ARCHIVE [MEMBER...]
xc32-ar -M [ <mri-script ]
```

12.6 OPTIONS

When you use the 32-bit archiver/librarian with command-line options, the archiver insists on at least two arguments to execute: one key letter specifying the operation (optionally accompanied by other key letters specifying modifiers), and the archive name.

```
xc32-ar [-]P[MOD [RELPOS][COUNT]] ARCHIVE [MEMBER...]
```

> **Note:** command-line options are case sensitive.

Most operations can also accept further *MEMBER* arguments, specifying archive members. Without specifying members, the entire archive is used.

The 32-bit archiver/librarian allows you to mix the operation code P and modifier flags *MOD* in any order, within the first command line argument. If you wish, you may begin the first command line argument with a dash.

The P key letter specifies what operation to execute; it may be any of the following, but you must specify only one of them.

TABLE 12-1: OPERATION TO EXECUTE

Option	Function
d	Delete modules from the archive. Specify the names of modules to be deleted as MEMBER...; the archive is untouched if you specify no files to delete. If you specify the v modifier, the 32-bit archiver/librarian lists each module as it is deleted.
m	Use this operation to move members in an archive. The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member. If no modifiers are used with m, any members you name in the MEMBER arguments are moved to the end of the archive; you can use the a, b or i modifiers to move them to a specified place instead.
p	Print the specified members of the archive, to the standard output file. If the v modifier is specified, show the member name before copying its contents to standard output. If you specify no MEMBER arguments, all the files in the archive are printed.
q	Append the files MEMBER... into ARCHIVE.
r	Insert the files MEMBER... into ARCHIVE (with replacement). If one of the files named in MEMBER... does not exist, the archiver displays an error message, and leaves undisturbed any existing members of the archive matching that name. By default, new members are added at the end of the file; but you may use one of the modifiers a, b or i to request placement relative to some existing member. The modifier v used with this operation elicits a line of output for each file inserted, along with one of the letters a or r to indicate whether the file was appended (no old member deleted) or replaced.
t	Display a table listing the contents of ARCHIVE, or those of the files listed in MEMBER..., that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group and size, you can request that by also specifying the v modifier. If you do not specify a MEMBER, all files in the archive are listed. For example, if there is more than one file with the same name (fie) in an archive (b.a), then xc32-ar t b.a fie lists only the first instance; to see them all, you must ask for a complete listing in xc32-ar t b.a.
x	Extract members (named MEMBER) from the archive. You can use the v modifier with this operation, to request that the archiver list each name as it extracts it. If you do not specify a MEMBER, all files in the archive are extracted.

A number of modifiers (MOD) may immediately follow the P key letter to specify variations on an operation's behavior.

TABLE 12-2: MODIFIERS

Option	Function
a	Add new files after an existing member of the archive. If you use the modifier a, the name of an existing archive member must be present as the RELPOS argument, before the ARCHIVE specification.
b	Add new files before an existing member of the archive. If you use the modifier b, the name of an existing archive member must be present as the RELPOS argument, before the ARCHIVE specification. (Same as i.)
c	Create the archive. The specified ARCHIVE is always created if it did not exist, when you requested an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.
f	Truncate names in the archive. The 32-bit archiver/librarian will normally permit file names of any length. This will cause it to create archives that are not compatible with the native archiver program on some systems. If this is a concern, the f modifier may be used to truncate file names when putting them in the archive.
i	Insert new files before an existing member of the archive. If you use the modifier i, the name of an existing archive member must be present as the RELPOS argument, before the ARCHIVE specification. (Same as b.)
l	This modifier is accepted but not used.
N	Uses the COUNT parameter. This is used if there are multiple entries in the archive with the same name. Extract or delete instance COUNT of the given name from the archive.
o	Preserve the original dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.
P	Use the full path name when matching names in the archive. The 32-bit archiver/librarian cannot create an archive with a full path name (such archives are not POSIX compliant), but other archive creators can. This option will cause the archiver to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.
s	Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running xc32-ar s on an archive is equivalent to running ranlib on it.
S	Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive cannot be used with the linker. In order to build a symbol table, you must omit the S modifier on the last execution of the archiver, or you must run ranlib on the archive.
u	Normally, xc32-ar r... inserts all files listed into the archive. If you would like to insert only those of the files you list that are newer than existing members of the same names, use this modifier. The u modifier is allowed only for the operation r (replace). In particular, the combination qu is not allowed, since checking the timestamps would lose any speed advantage from the operation q.
v	This modifier requests the verbose version of an operation. Many operations display additional information, such as file names processed when the modifier v is appended.
V	This modifier shows the version number of the 32-bit archiver/librarian.

12.7 SCRIPTS

If you use the single command line option `-M` with the archiver, you can control its operation with a rudimentary command language.

```
xc32-ar -M [ <SCRIPT ]
```

This form of the 32-bit archiver/librarian operates interactively if standard input is coming directly from a terminal. During interactive use, the archiver prompts for input (the prompt is AR >), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and the 32-bit archiver/librarian abandons execution (with a nonzero exit code) on any error.

The archiver command language is **not** designed to be equivalent to the command-line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to the 32-bit archiver/librarian for developers who already have scripts written for the MRI "librarian" program.

The syntax for the 32-bit archiver/librarian command language is straightforward:

- commands are recognized in upper or lower case; for example, `LIST` is the same as `list`. In the following descriptions, commands are shown in upper case for clarity.
- a single command may appear on each line; it is the first word on the line.
- empty lines are allowed, and have no effect.
- comments are allowed; text after either of the characters "*" or ";" is ignored.
- Whenever you use a list of names as part of the argument to an `xc32-ar` command, you can separate the individual names with either commas or blanks. Commas are shown in the explanations below, for clarity.
- "+" is used as a line continuation character; if "+" appears at the end of a line, the text on the following line is considered part of the current command.

Table shows the commands you can use in archiver scripts, or when using the archiver interactively. Three of them have special significance.

ARCHIVER SCRIPTS COMMANDS

Option	Function
`OPEN` or `CREATE`	Specify a "current archive", which is a temporary file required for most of the other commands.
`SAVE`	Commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.
`ADDLIB ARCHIVE` `ADDLIB ARCHIVE (MODULE, MODULE,...MODULE)`	Add all the contents of `ARCHIVE` (or, if specified, each named `MODULE` from `ARCHIVE`) to the current archive. Requires prior use of `OPEN` or `CREATE`.
`ADDMOD MEMBER, MEMBER, ... MEMBER`	Add each named `MEMBER` as a module in the current archive. Requires prior use of `OPEN` or `CREATE`.
`CLEAR`	Discard the contents of the current archive, canceling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.

ARCHIVER SCRIPTS COMMANDS (CONTINUED)

Option	Function
CREATE ARCHIVE	Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as ARCHIVE until you use SAVE. You can overwrite existing archives; similarly, the contents of any existing file named ARCHIVE will not be destroyed until SAVE.
DELETE MODULE, MODULE, ... MODULE	Delete each listed MODULE from the current archive; equivalent to xc32-ar -d ARCHIVE MODULE ... MODULE. Requires prior use of OPEN or CREATE.
DIRECTORY ARCHIVE (MODULE, ... MODULE) [OUTPUTFILE]	List each named MODULE present in ARCHIVE. The separate command VERBOSE specifies the form of the output: when verbose output is off, output is like that of xc32-ar -t ARCHIVE MODULE.... When verbose output is on, the listing is like xc32-ar -tv ARCHIVE MODULE.... Output normally goes to the standard output stream; however, if you specify OUTPUTFILE as a final argument, the 32-bit archiver/librarian directs the output to that file.
END	Exit from the archiver with a 0 exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last SAVE command, those changes are lost.
EXTRACT MODULE, MODULE, ... MODULE	Extract each named MODULE from the current archive, writing them into the current directory as separate files. Equivalent to xc32-ar -x ARCHIVE MODULE.... Requires prior use of OPEN or CREATE.
LIST	Display full contents of the current archive, in “verbose” style regardless of the state of VERBOSE. The effect is like xc32-ar tv ARCHIVE. (This single command is a 32-bit archiver/librarian enhancement, rather than present for MRI compatibility.) Requires prior use of OPEN or CREATE.
OPEN ARCHIVE	Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect ARCHIVE until you next use SAVE.
REPLACE MODULE, MODULE, ... MODULE	In the current archive, replace each existing MODULE (named in the REPLACE arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist. Requires prior use of OPEN or CREATE.
VERBOSE	Toggle an internal flag governing the output from DIRECTORY. When the flag is on, DIRECTORY output matches output from xc32-ar -tv ....
SAVE	Commits your changes to the current archive and actually saves it as a file with the name specified in the last CREATE or OPEN command. Requires prior use of OPEN or CREATE.

NOTES:

Chapter 13. Other Utilities

13.1 INTRODUCTION

Besides the MPLAB XC32 Object Archiver/Librarian (xc32-ar), there are several other binary utilities available for use with the MPLAB XC32 Assembler and Linker.

Topics covered in this chapter are:

Utility	Description
xc32-bin2hex Utility	Converts a linked object file into an Intel® hex file.
xc32-nm Utility	Lists symbols from an object file.
xc32-objdump Utility	Displays information about object files.
xc32-ranlib Utility	Generates an index from the contents of an archive and stores it in the archive.
xc32-size Utility	List file section sizes and total size.
xc32-strings Utility	Prints the printable character sequences.
xc32-strip Utility	Discards all symbols from an object file.

13.2 xc32-bin2hex UTILITY

The binary-to-hexadecimal (`xc32-bin2hex`) utility converts binary files (from the 32-bit linker) to Intel hex format files, suitable for loading into device programmers.

- Input/Output Files
- Syntax
- Options

13.2.1 Input/Output Files

- Input: ELF formatted binary object files
- Output: Intel hex files

13.2.2 Syntax

Command line syntax is:

`xc32-bin2hex [options] file`

Example 13.1: hello.elf

Convert the absolute ELF executable file `hello.elf` to `hello.hex`

```
xc32-bin2hex hello.elf
```

13.2.3 Options

The following options are supported.

TABLE 13-1: xc32-bin2hex OPTIONS

Option	Function
`-a, --sort`	Sort sections by address
`-i, --virtual`	Use virtual addresses
`-p, --physical`	Use physical addresses **(default)**
`-v, --verbose`	Print verbose messages
`-?, --help`	Print a help screen

Note: See the *PIC32MX Family Reference Manual* (DS61115) for a description of the PIC32MX Virtual-to-Physical Fixed Memory Mapping.

EXAMPLE 13-2: -v OPTION OUTPUT

```
writing hello.hex

section  PC address   byte address   length (w/pad)   actual length  (dec)
-------   ----------   ------------   --------------   --------------------
.reset             0              0              0x8             0x6  (6)
.text          0x100          0x200           0x6a28          0x4f9e  (20382)
.dinit        0x3614         0x6c28            0xda4           0xa3b  (2619)
.const        0x3ce6         0x79cc             0x40            0x30  (48)
.ivt             0x4            0x8             0xf8            0xba  (186)
.aivt           0x84          0x108             0xf8            0xba  (186)

Total program memory used (bytes):                                0x5b83  (23427)
```

13.3 xc32-nm UTILITY

The `xc32-nm` utility produces a list of symbols from object files. Each item in the list consists of the symbol value, symbol type and symbol name.

- Input Files
- Syntax
- Options
- Output Formats

13.3.1 Input Files

- Input: ELF object file. If no object files are listed as arguments, `xc32-nm` assumes the file `a.out`.

13.3.2 Syntax

Command line syntax is:

```
xc32-nm [ -A | -o | --print-file-name ]
      [ -a | --debug-syms ] [ -B ]
      [ --defined-only ] [ -u | --undefined-only ]
      [ -f format | --format=format ] [ -g | --extern-only ]
      [ --help ] [-l | --line-numbers ]
      [ -n | -v | --numeric-sort ]  [-omf=format]
      [ -p | --no-sort ]
      [ -P | --portability ] [ -r | --reverse-sort ]
      [ -s  --print-armap ] [ --size-sort ]
      [ -t radix | --radix=radix ] [ -V | --version ]
      [ OBJFILE... ]
```

13.3.3 Options

The long and short forms of options, shown in Table 13-2 as alternatives, are equivalent.

TABLE 13-2: xc32-nm OPTIONS

Option	Function
-A -o --print-file-name	Precede each symbol by the name of the input file (or archive member) in which it was found, rather than identifying the input file once only, before all of its symbols.
-a --debug-syms	Display all symbols, even debugger-only symbols; normally these are not listed.
-B	The same as --format=bsd.
--defined-only	Display only defined symbols for each object file.
-u --undefined-only	Display only undefined symbols (those external to each object file).
-f *format* --format=*format*	Use the output format *format*, which can be bsd, sysv or posix. The default is bsd. Only the first character of *format* is significant; it can be either upper or lower case.
-g --extern-only	Display only external symbols.
--help	Show a summary of the options to xc32-nm and exit.
-l --line-numbers	For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry that refers to the symbol. If line number information can be found, print it after the other symbol information.
-n -v --numeric-sort	Sort symbols numerically by their addresses, rather than alphabetically by their names.
-p --no-sort	Do not bother to sort the symbols in any order; print them in the order encountered.
-P --portability	Use the POSIX.2 standard output format instead of the default format. Equivalent to -f posix.
-r --reverse-sort	Reverse the order of the sort (whether numeric or alphabetic); let the last come first.
-s --print-armap	When listing symbols from archive members, include the index: a mapping (stored in the archive by xc32-ar or xc32-ranlib) of which modules contain definitions for which names.
--size-sort	Sort symbols by size. The size is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.
-t *radix* --radix=*radix*	Use *radix* as the radix for printing the symbol values. It must be d for decimal, o for octal or x for hexadecimal.
-V --version	Show the version number of xc32-nm and exit.

13.3.4 Output Formats

The symbol value is in the radix selected by the options, or hexadecimal by default.

If the symbol type is lowercase, the symbol is local; if uppercase, the symbol is global (external). Table 13-3 shows the symbol types:

TABLE 13-3: SYMBOL TYPES

Symbol	Description
A	The symbol's value is absolute, and will not be changed by further linking.
B	The symbol is in the uninitialized data section (known as BSS).
C	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
D	The symbol is in the initialized data section.
N	The symbol is a debugging symbol.
R	The symbol is in a read only data section.
T	The symbol is in the text (code) section.
U	The symbol is undefined.
V	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
W	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
?	The symbol type is unknown, or object file format specific.

13.4 xc32-objdump UTILITY

The `xc32-objdump` utility displays information about one or more object files. The options control what particular information to display. The output can provide information similar to that of a disassembly listing.

- Input Files
- Syntax
- Options

13.4.1 Input Files

- Input: Object archive files. If no object files are listed as arguments, `xc32-objdump` assumes the file `a.out`.

13.4.2 Syntax

Command line syntax is:

```
xc32-objdump [ -a | --archive-headers ]
      [ -d | --disassemble ]
      [ -D | --disassemble-all ]
      [ -f | --file-headers ]
      [ --file-start-context ]
      [ -g | --debugging ]
      [ -h | --section-headers | --headers ]
      [ -H | --help ]
      [ -j name | --section=name ]
      [ -l | --line-numbers ]
      [ -M options | --disassembler-options=options]
      [ --prefix-addresses]
      [ -r | --reloc ]
      [ -s | --full-contents ]
      [ -S | --source ]
      [ --[no-]show-raw-insn ]
      [ --start-address=address ]
      [ --stop-address=address ]
      [ -t | --syms ]
      [ -V | --version ]
      [ -w | --wide ]
      [ -x | --all-headers ]
      [ -z | --disassemble-zeros ]
      OBJFILE...
```

`OBJFILE…` are the object files to be examined. When you specify archives, `xc32-objdump` shows information on each of the member object files.

13.4.3 Options

The long and short forms of options, shown in Table 13-4, as alternatives, are equivalent. At least one of the following options `-a, -d, -D, -f, -g, -G, -h, -H, -p, -r, -R, -S, -t, -T, -V` or `-x` must be given.

TABLE 13-4: xc32-objdump OPTIONS

Option	Function
`-a` `--archive-header`	If any of the `OBJFILE` files are archives, display the archive header information (in a format similar to `ls -l`). Besides the information you could list with `xc32-ar tv`, `xc32-objdump -a` shows the object file format of each archive member.
`-d` `--disassemble`	Display the assembler mnemonics for the machine instructions from `OBJFILE`. This option only disassembles those sections that are expected to contain instructions.
`-D` `--disassemble-all`	Like `-d`, but disassemble the contents of all sections, not just those expected to contain instructions.
`-f` `--file-header`	Display summary information from the overall header of each of the `OBJFILE` files.
`--file-start-context`	Specify that when displaying inter-listed source code/disassembly (assumes `'-S'`) from a file that has not yet been displayed, extend the context to the start of the file.
`-g` `--debugging`	Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C like syntax. Only certain types of debugging information have been implemented.

`-h`
`--section-header`
`--header`
	Display summary information from the section headers of the object file.
	File segments may be relocated to nonstandard addresses, for example by using the `-Ttext, -Tdata` or `-Tbss` options to `ld`. However, some object file formats, such as `a.out`, do not store the starting address of the file segments. In those situations, although `ld` relocates the sections correctly, using `xc32-objdump -h` to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.

`-H` `--help`	Print a summary of the options to `xc32-objdump` and exit.
`-j` *name* `--section=`*name*	Display information only for section *name*.
`-l` `--line-numbers`	Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with `-d, -D` or `-r`.
`-M` *options* `--disassembler-` `options=`*options*	Pass target specific information to the disassembler. The PIC32 device supports the following target specific options: `symbolic` - Will perform symbolic disassembly.
`--prefix-addresses`	When disassembling, print the complete address on each line. This is the older disassembly format.
`-r` `--reloc`	Print the relocation entries of the file. If used with `-d` or `-D`, the relocations are printed interspersed with the disassembly.
`-s` `--full-contents`	Display the full contents of any sections requested.

TABLE 13-4: xc32-objdump OPTIONS (CONTINUED)

Option	Function
-S --source	Display source code intermixed with disassembly, if possible. Implies -d.
--show-raw-insn	When disassembling instructions, print the instruction in hex, as well as in symbolic form. This is the default except when --prefix-addresses is used.
--no-show-raw-insn	When disassembling instructions, do not print the instruction bytes. This is the default when --prefix-addresses is used.
--start-address=*address*	Start displaying data at the specified address. This affects the output of the -d, -r and -s options.
--stop-address=*address*	Stop displaying data at the specified address. This affects the output of the -d, -r and -s options.
-t --syms	Print the symbol table entries of the file. This is similar to the information provided by the xc32-nm program.
-V --version	Print the version number of xc32-objdump and exit.
-w --wide	Format some lines for output devices that have more than 80 columns.
-x --all-header	Display all available header information, including the symbol table and relocation entries. Using -x is equivalent to specifying all of -a -f -h -r -t.
-z --disassemble-zeroes	Normally the disassembly output will skip blocks of zeros. This option directs the disassembler to disassemble those blocks, just like any other data.

13.5 xc32-ranlib UTILITY

The `xc32-ranlib` utility generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use `xc32-nm -s` or `xc32-nm --print-armap` to list this index. An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

Running `xc32-ranlib` is completely equivalent to executing `xc32-ar -s` (i.e., the 32-bit archiver/librarian with the `-s` option).

- Input/Output Files
- Syntax
- Options

13.5.1 Input/Output Files

- Input: Archive file
- Output: Archive file

13.5.2 Syntax

Command line syntax is:

```
xc32-ranlib [-v | -V | --version] ARCHIVE
xc32-ranlib [-h | --help]
```

13.5.3 Options

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

TABLE 13-5: xc32-ranlib OPTIONS

Option	Function

`-v`
`-V`
`--version`
	Show the version number of `xc32-ranlib`

`-h` `--help`	Print a help message

13.6 xc32-size UTILITY

The `xc32-size` utility lists the section sizes, and the total size, for each of the object or archive files in its argument list. By default, one line of output is generated for each object file or each module in an archive.

> **Note:** The linker's `--report-mem` memory-usage report provides additional information on memory usage.

- Input/Output Files
- Syntax
- Options
- Example

13.6.1 Input/Output Files

- Input: Object or archive file(s)
- Output: Standard output

13.6.2 Syntax

The `xc32-size` command-line syntax is:

```
xc32-size [ -A | -B | --format=compatibility ]
[ --help ]
[ -d | -o | -x | --radix= number]
[ -t | --totals ]
[ -V | --version ]
[objfile...]
```

13.6.3 Options

The `xc32-size` options are shown below.

TABLE 13-6: xc32-size OPTIONS

Option	Function

`-A`
`-B`
`--format=`*compatibility*
	Using one of these options, you can choose whether the output from gnu size resembles output from System V size (using `-A` or `--format=sysv`), or Berkeley size (using `-B` or `--format=berkeley`). The default is the one-line format similar to Berkeley's.

`--help`	Show a summary of acceptable arguments and options.

`-d`
`-o`
`-x`
`--radix=`*number*
	Using one of these options, you can control whether the size of each section is given in decimal (`-d` or `--radix=10`); octal (`-o` or `--radix=8`); or hexadecimal (`-x` or `--radix=16`). In `--radix=`*number*, only the three values (8, 10, 16) are supported. The total size is always given in two radices; decimal and hexadecimal for `-d` or `-x` output, or octal and hexadecimal if you're using `-o`.

`-t` `--totals`	Show totals of all objects listed (Berkeley format listing mode only).
`-V` `--version`	Display the version number of `xc32-size`.

13.6.4 Example

Here is an example of the Berkeley (default) format of output from size:

```
xc32-size --format=Berkeley ranlib size
text   data  bss   dec    hex   filename
294880 81920 11592 388392 5ed28 ranlib
294880 81920 11888 388688 5ee50 size
```

This is the same data, but displayed closer to System V conventions:

```
xc32-size --format=SysV ranlib size
ranlib :
section   size   addr
.text   294880   8192
.data    81920 303104
.bss     11592 385024

Total 388392
size :
section   size   addr
.text   294880   8192
.data    81920 303104
.bss     11888 385024
Total 388688
```

13.7 xc32-strings UTILITY

For each file given, the `xc32-strings` utility prints the printable character sequences that are at least 4 characters long (or the number given in the options) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

`xc32-strings` is mainly useful for determining the contents of non-text files.

- Input/Output Files
- Syntax
- Options

13.7.1 Input/Output Files

- Input: ELF object file
- Output: Standard output

13.7.2 Syntax

Command line syntax is:

```
xc32-strings [-a | --all | -] [-f | --print-file-name]
      [--help] [-min-len | -n min-len | --bytes=min-len]
      [-t radix | --radix=radix] [-v | --version] FILE...
```

13.7.3 Options

The long and short forms of options, shown in Table 13-7 as alternatives, are equivalent.

TABLE 13-7: xc32-strings OPTIONS

Option	Function

`-a`
`--all`
`-`
	Do not scan only the initialized and loaded sections of object files; scan the whole files.

`-f` `--print-file-name`	Print the name of the file before each string.
`--help`	Print a summary of the program usage on the standard output and exit.
`-min-len` `-n` *`min-len`* `--bytes=`*`min-len`*	Print sequences of characters that are at least *`-min-len`* characters long, instead of the default 4.
`-t` *`radix`* `--radix=`*`radix`*	Print the offset within the file before each string. The single character argument specifies the radix of the offset - `o` for octal, `x` for hexadecimal or `d` for decimal.
`-v` `--version`	Print the program version number on the standard output and exit.

13.8 xc32-strip UTILITY

The `xc32-strip` utility discards all symbols from the object and archive files specified. At least one file must be given. `xc32-strip` modifies the files named in its argument, rather than writing modified copies under different names.

- Input/Output Files
- Syntax
- Options

13.8.1 Input/Output Files

- Input: Object or archive files
- Output: Object or archive files. If no object or archive files are listed as arguments, `xc32-strip` assumes the file `a.out`.

13.8.2 Syntax

Command line syntax is:

```
xc32-strip [ -g | -S | --strip-debug ] [ --help ]
      [ -K symbolname | --keep-symbol=symbolname ]
      [ -N symbolname | --strip-symbol=symbolname ]
      [ -o file ]
      [ -p | --preserve-dates ]
      [ -R sectionname | --remove-section=sectionname ]
      [ -s | --strip-all ] [--strip-unneeded]
      [ -v | --verbose ]  [ -V | --version ]
      [ -x | --discard-all ] [ -X | --discard-locals ]
      OBJFILE...
```

13.8.3 Options

The long and short forms of options, shown in Table 13-8 as alternatives, are equivalent.

TABLE 13-8: xc32-strip OPTIONS

Option	Function
-g -S --strip-debug	Remove debugging symbols only.
--help	Show a summary of the options to xc32-strip and exit.
-K *symbolname* --keep-symbol=*symbolname*	Keep only symbol *symbolname* from the source file. This option may be given more than once.
-N *symbolname* --strip-symbol=*symbolname*	Remove symbol *symbolname* from the source file. This option may be given more than once, and may be combined with strip options other than -K.
-o *file*	Put the stripped output in *file*, rather than replacing the existing file. When this argument is used, only one OBJFILE argument may be specified.
-p --preserve-dates	Preserve the access and modification dates of the file.
-R *sectionname* --remove-section=*sectionname*	Remove any section named *sectionname* from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable.
-s --strip-all	Remove all symbols.
--strip-unneeded	Remove all symbols that are not needed for relocation processing.
-v --verbose	Verbose output: list all object files modified. In the case of archives, xc32-strip -v lists all members of the archive.
-V --version	Show the version number for xc32-strip.
-x --discard-all	Remove non-global symbols.
-X --discard-locals	Remove compiler-generated local symbols. (These usually start with L or ".".)

Part 4 – Appendices

NOTES:

Appendix A. Deprecated Features

A.1 INTRODUCTION

The features described below are considered to be obsolete and have been replaced with more advanced functionality. Projects that depend on deprecated features will work properly with versions of the language tools cited. The use of a deprecated feature will result in a warning; programmers are encouraged to revise their projects in order to eliminate any dependency on deprecated features. Support for these features may be removed entirely in future versions.

A.2 ASSEMBLER DIRECTIVES THAT DEFINE SECTIONS

The following `.section` directive format was deprecated in XC32 v2.00. The new directive format may be found in **Section 4.2 "Directives that Define Sections"**.

.section *name* [, *flags*] [, @*type*]

Use the `.section` directive to assemble the following code into a section named `name`. The optional flags argument is a quoted string which may contain any combination of the following characters:

`a` section is allocatable

`w` section is writable

`x` section is executable

The @*type* argument may be one of:

@`progbits` Normal section with contents

@`nobits` Section does not contain data (i.e., section only occupies space)

The following section names are recognized:

TABLE A-1: SECTION NAMES

Section Name	Default Flag
.text	x
.data	d
.bss	b

Note: Ensure that double quotes are used around flags. If the optional argument to the `.section` directive is not quoted, it is taken as a sub-section number. Remember, a single character in single quotes (i.e., 'b') is converted by the preprocessor to a number.

Section Directive Examples

```
.section foo,"aw",@progbits #foo is initialized
#data memory.
.section fob,"aw",@nobits #fob is uninitialized
#(but also not zeroed)
#data memory.
.section bar,"ax",@progbits #bar is in program memory
```

Appendix B. Useful Tables

B.1 INTRODUCTION

Here are some useful tables included for reference.

- ASCII Character Set
- Hexadecimal to Decimal Conversion

B.2 ASCII CHARACTER SET

This table lists the ASCII standard character set.

	Most Significant Nibble								
	HEX	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Least Significant Nibble	**0**	NUL	DLE	Space	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	–	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

B.3 HEXADECIMAL TO DECIMAL CONVERSION

This table describes how to convert hexadecimal to decimal. For each hex digit, find the associated decimal value. Add the numbers together.

High Byte				Low Byte			
Hex 1000	**Dec**	**Hex 100**	**Dec**	**Hex 10**	**Dec**	**Hex 1**	**Dec**
0	0	0	0	0	0	0	0
1	4096	1	256	1	16	1	1
2	8192	2	512	2	32	2	2
3	12288	3	768	3	48	3	3
4	16384	4	1024	4	64	4	4
5	20480	5	1280	5	80	5	5
6	24576	6	1536	6	96	6	6
7	28672	7	1792	7	112	7	7
8	32768	8	2048	8	128	8	8
9	36864	9	2304	9	144	9	9
A	40960	A	2560	A	160	A	10
B	45056	B	2816	B	176	B	11
C	49152	C	3072	C	192	C	12
D	53248	D	3328	D	208	D	13
E	57344	E	3584	E	224	E	14
F	61440	F	3840	F	240	F	15

For example, hex A38F converts to 41871 as follows:

Hex 1000's Digit	Hex 100's Digit	Hex 10's Digit	Hex 1's Digit	Result
40960	768	128	15	41871 Decimal

Appendix C. GNU Free Documentation License

NOTES:

Glossary

A

Absolute Section

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

Absolute Variable/Function

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

Access Memory

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

Access Entry Points

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

Address

Value that identifies a location in memory.

Alphabetic Character

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

Alphanumeric

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

ANDed Breakpoints

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

Anonymous Structure

16-bit C Compiler **–** An unnamed structure.

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

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

ANSI

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

Application

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

Archive/Archiver

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

ASCII

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

Assembly/Assembler

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

Assigned Section

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

Asynchronously

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

Asynchronous Stimulus

Data generated to simulate external inputs to a simulator device.

Attribute

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

Attribute, Section

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

B

Binary

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

Bookmarks

Use bookmarks to easily locate specific lines in a file.

Select Toggle Bookmarks on the Editor toolbar to add/remove bookmarks. Click other icons on this toolbar to move to the next or previous bookmark.

Breakpoint

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

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

Build

Compile and link all the source files for an application.

C

C\C++

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

Calibration Memory

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

Central Processing Unit

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

Clean

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

COFF

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

Command Line Interface

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

Compiled Stack

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

Compiler

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

Conditional Assembly

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

Conditional Compilation

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

Configuration Bits

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

Control Directives

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

CPU

See Central Processing Unit.

Cross Reference File

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

D

Data Directives

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

Data Memory

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

Data Monitor and Control Interface (DMCI)

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

Debug/Debugger

See ICE/ICD.

Debugging Information

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

Deprecated Features

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

Device Programmer

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

Digital Signal Controller

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

Digital Signal Processing\Digital Signal Processor

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

Directives

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

Download

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

DWARF

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

E

EEPROM

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

ELF

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

Emulation/Emulator

See ICE/ICD.

Endianness

The ordering of bytes in a multi-byte object.

Environment

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

Epilogue

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

EPROM

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

Error/Error File

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

Event

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

Executable Code

Software that is ready to be loaded for execution.

Export

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

Expressions

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

Extended Microcontroller Mode

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

Extended Mode (PIC18 MCUs)

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

External Label

A label that has external linkage.

External Linkage

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

External Symbol

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

External Symbol Resolution

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

External Input Line

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

External RAM

Off-chip Read/Write memory.

F

Fatal Error

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

File Registers

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

Filter

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

Fixup

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

Flash

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

FNOP

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

Frame Pointer

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

Free-Standing

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

G

GPR

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

H

Halt

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

Heap

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

Hex Code\Hex File

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

Hexadecimal

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

High Level Language

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

I

ICE/ICD

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

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

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

ICSP

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

IDE

Integrated Development Environment, as in MPLAB IDE/MPLAB X IDE.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

Import

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

Initialized Data

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

```
int myVar=5;
```

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

Instruction Set

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

Instructions

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

Internal Linkage

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

International Organization for Standardization

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

Interrupt

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

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Service Request (IRQ)

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

Interrupt Service Routine (ISR)

Language tools – A function that handles an interrupt.

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

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

L

L-value

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

Latency

The time between an event and its response.

Library/Librarian

See Archive/Archiver.

Linker

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

Linker Script Files

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

Listing Directives

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

Listing File

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

Little Endian

A data ordering scheme for multibyte data whereby the least significant byte is stored at the lower addresses.

Local Label

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

Logic Probes

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

Loop-Back Test Board

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

LVDS

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

With standard I/0 signaling, data storage is contingent upon the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

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

M

Machine Code

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

Machine Language

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

Macro

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

Macro Directives

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

Makefile

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

Make Project

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

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also uC.

Memory Model

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

Message

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

Microcontroller

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

Microcontroller Mode

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

Microprocessor Mode

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

Mnemonics

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

Module

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

MPASM™ Assembler

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

MPLAB *Language Tool* for *Device*

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

MPLAB ICD

Microchip in-circuit debugger that works with MPLAB IDE/MPLAB X IDE. *See* ICE/ICD.

MPLAB IDE/MPLAB X IDE

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

MPLAB PM3

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

MPLAB REAL ICE™ In-Circuit Emulator

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

MPLAB SIM

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

MPLIB™ Object Librarian

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

MPLINK™ Object Linker

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

MRU

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

N

Native Data Size

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

Nesting Depth

The maximum level to which macros can include other macros.

Node

MPLAB IDE/MPLAB X IDE project component.

Non-Extended Mode (PIC18 MCUs)

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

Non Real Time

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

Non-Volatile Storage

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

NOP

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

O

Object Code/Object File

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

Object File Directives

Directives that are used only when creating an object file.

Octal

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

Off-Chip Memory

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

Opcodes

Operational Codes. *See* Mnemonics.

Operators

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

OTP

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

P

Pass Counter

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

PC

Personal Computer or Program Counter.

PC Host

Any PC running a supported Windows operating system.

Persistent Data

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

Phantom Byte

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

PIC MCUs

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

PICkit 2 and 3

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

Plug-ins

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

Pod

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

Power-on-Reset Emulation

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

Pragma

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

Precedence

Rules that define the order of evaluation in expressions.

Production Programmer

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

Profile

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

Program Counter

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

Program Counter Unit

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

Program Memory

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

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

Project

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

Prologue

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

Prototype System

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

Psect

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

PWM Signals

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

Q

Qualifier

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

R

Radix

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

RAM

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

Raw Data

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

Read Only Memory

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

Real Time

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

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

Recursive Calls

A function that calls itself, either directly or indirectly.

Recursion

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

Reentrant

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

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. MPLAB XC32 currently knows how to `relax` a `CALL` instruction into an `RCALL` instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

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

Relocatable Section

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

Relocation

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

ROM

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

Run

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

Run-time Model

Describes the use of target architecture resources.

Runtime Watch

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

S

Scenario

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

Section

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

Section Attribute

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

Sequenced Breakpoints

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

Serialized Quick Turn Programming

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

Shell

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

Simulator

A software program that models the operation of devices.

Single Step

This command steps though code, one instruction at a time. After each instruction, MPLAB IDE/MPLAB X 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/MPLAB X IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew

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

Skid

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

Source Code

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

Source File

An ASCII text file containing source code.

Special Function Registers (SFRs)

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

SQTP

See Serialized Quick Turn Programming.

Stack, Hardware

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

Stack, Software

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

Stack, Compiled

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

MPLAB Starter Kit for *Device*

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

Static RAM or SRAM

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

Status Bar

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

Step Into

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

Step Over

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

Step Out

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

Stimulus

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

Stopwatch

A counter for measuring execution cycles.

Storage Class

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

Storage Qualifier

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

Symbol

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

Symbol, Absolute

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

System Window Control

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

T

Target

Refers to user hardware.

Target Application

Software residing on the target board.

Target Board

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

Target Processor

The microcontroller device on the target application board.

Template

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

Tool Bar

A row or column of icons that you can click on to execute MPLAB IDE/MPLAB X 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/MPLAB X IDE's trace window.

Trace Memory

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

Trace Macro

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

Trigger Output

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

Trigraphs

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

U

Unassigned Section

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

Uninitialized Data

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

```
int myVar;
```

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

Upload

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

USB

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

V

Vector

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

Volatile

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

W

Warning

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

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

Watch Variable

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

Watch Window

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

Watchdog Timer (WDT)

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

Workbook

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

NOTES:

Index

Symbols

N

O

NOTES: