MPLAB® XC32 Assembler, Linker and Utilities User's Guide

Notice to Development Tools Customers

Important:
All documentation becomes dated, and Development Tools manuals are no exception. Our 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 website (www.microchip.com/) to obtain the latest version of the PDF document.

Documents are identified with a DS number located on the bottom of each page. The DS format is DS<DocumentNumber><Version>, where <DocumentNumber> is an 8-digit number and <Version> is an uppercase letter.

For the most up-to-date information, find help for your tool at onlinedocs.microchip.com/.

Table of Contents

1. Preface

MPLAB® XC32 Assembler, Linker and Utilities and support information is discussed in this user's guide.

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® XC32 Assembler, Linker and Utilities 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 File and then Save.
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
N'Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4'b0010, 2'hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain Courier New	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`C:\Users\User1\Projects`
	Keywords	`static`, `auto`, `extern`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier New	A variable argument	*`file.o`*, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`xc8 [`*`options`*`]` *`files`*
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)`
		`{ ...`
		`}`

1.2 Recommended Reading

This document describes how to use the MPLAB 32-bit assembler, object linker, object archiver/librarian and various utilities. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Release Notes (Readme Files)

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

MPLAB® XC32 C/C++ Compiler User's Guide for PIC32C/SAM MCUs (DS50002895) and **MPLAB® XC32 C/C++ Compiler User's Guide for PIC32M MCUs (DS50002799)**

These guides provide information on the compilers for the PIC32C/SAM and PIC32M MCUs, respectively.

Microchip Unified Standard Library Reference Guide (DS50003209)

Lists all standard C library functions provided with the MPLAB XC32 C/C++ Compiler with detailed descriptions of their use.

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

2. Assembler Overview

2.1 Introduction

The MPLAB® XC32 Assembler (`xc32-as`) produces relocatable machine code from symbolic assembly language for most XC32 supported Microchip devices.

The assembler is a console application for developing assembly language code and is available for Windows®, Linux® and macOS® operating systems.

The assembler is a port of the GNU assembler from the Free Software Foundation.

2.2 Assembler and Other Development Tools

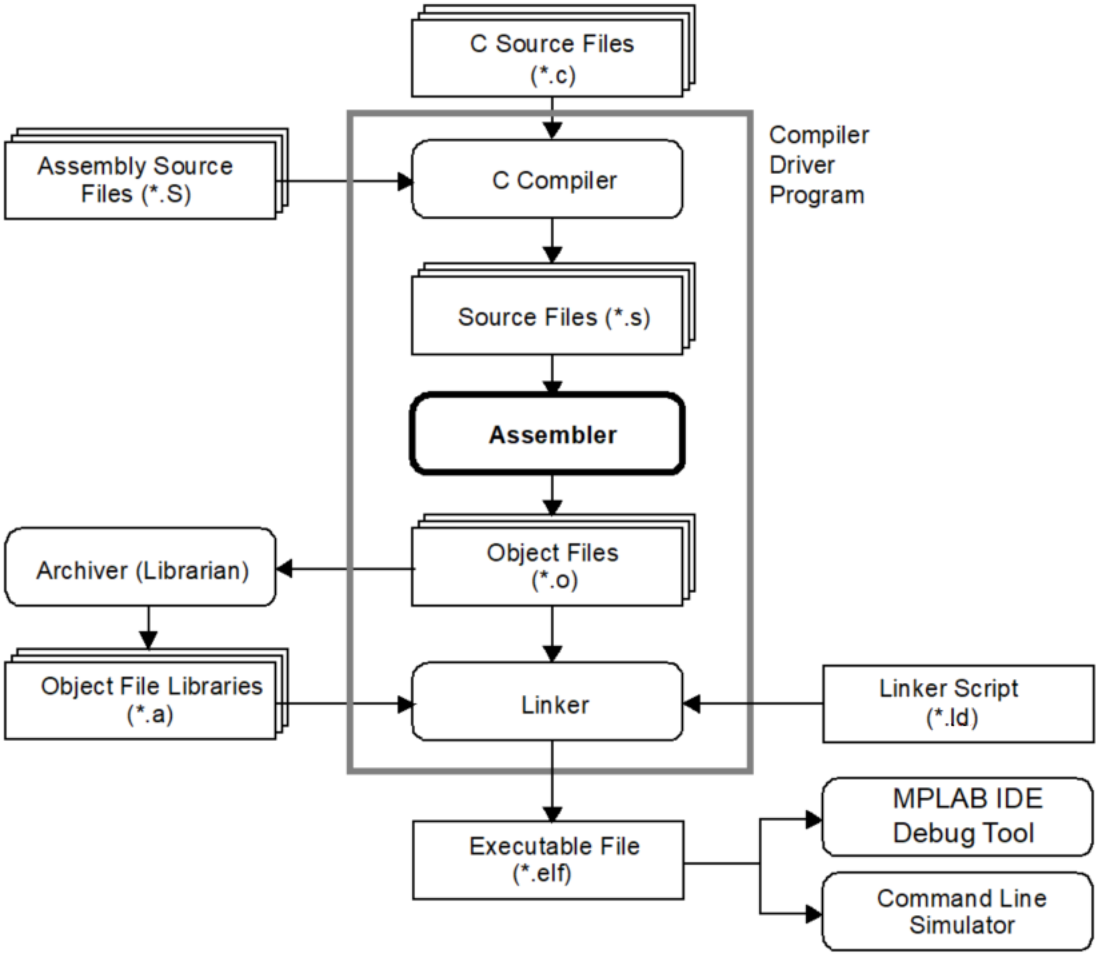
The MPLAB XC32 Assembler translates hand-written assembly source files. In addition, the MPLAB XC32 C/C++ Compiler uses this assembler to produce object files for the linker.

Assembly source files can contain C preprocessor directives. Such files must use a `.S` (capital 'S') extension and will be preprocessed before being passed to the assembler. Source files containing pure assembler code can use the `.s` extension (lower case 's') and will not be preprocessed.

A relocatable object file is generated by the assembler for each assembly source file, which can then be placed into a library archive or linked with other relocatable object files and/or archives to create an executable file.

The figure below shows the assembler's role in the compilation process.

Figure 2-1. Assembler Execution in the Compilation Process



2.3 Feature Set

Notable features of the assembler include:

- Support for the MIPS32, MIPS16e, and microMIPS, as well as ARM, Thumb, and Thumb-2 instruction sets
- Support for ELF object formats
- Available for Windows®, Linux® and macOS® operating systems
- Rich assembler directive set
- Flexible macro language
- Command-Line interface
- Integrated component of MPLAB X IDE®

2.4 Input/Output Files

The standard assembler input and output files are listed below.

Extension	Description
Input	
`.S`	Assembly source file to be preprocessed (recommended)
`.s`	Assembly source file

...........continued

Extension	Description
Output	
`.o`	Object file
`.lst`	Assembler list file

The MPLAB XC32 Assembler does not generate error files, hex files, or symbol and debug files. The XC32 assembler is capable of creating an assembler list file and a relocatable object file (that might or might not contain debugging information). The XC32 Linker 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 2.2. Assembler and Other Development Tools).

2.4.1 Source Files

The assembler accepts, as input, a source file that consists of either MIPS- or ARM-based instructions, assembler directives and comments. A sample source file is shown below.

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 relevant MPLAB XC32 compiler user's guide for more details on the C compiler driver.

Example 2-1. 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
```

2.4.2 Object File

The assembler creates a relocatable ELF object file for each source file processed. These object files do not have addresses resolved and must be linked with the other object files or library archives in the project to produce a final executable.

If the name of the output file is not specified with an option, the output file is called `a.out`. Use the `-o` command-line option (see 3. Assembler Command-Line Options) to override the default name.

2.4.3 Listing File

The assembler has the capability to produce an assembler list file. Since the list file is produced before the linker has executed, not all the addresses that appear in the list file will be absolute. Many will be relative to a section.

By default, the list file is displayed on the standard output. Specify the `-a=`*`file`* option (see 3. Assembler Command-Line Options) on the command line to send the output to a list file with the specified name.

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

2.4.3.1 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.

2.4.3.2 Title

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

2.4.3.3 Subtitle

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

2.4.3.4 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>
```

2.4.3.5 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
```

Notes:

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.

2.4.3.6 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 2-2. 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 mPORTAClearBitsfoo.S:38 .text:0000000c endless
```

3. Assembler Command-Line Options

3.1 Introduction

The MPLAB XC32 Assembler (`xc32-as`) may be used on the host's command-line interface (e.g., `cmd.exe`) if you need to build projects written only in assembly code. When building C-based projects, the assembler is called by the `xc32-gcc` compilation driver, which can be integrated into the MPLAB X IDE project manager.

Many of the commonly used assembler options are accessible as controls in the MPLAB X IDE Project Properties dialog. However, for a more advanced option, you may have to specify the option in the “Alternate Settings” field of this dialog. After you build a project in MPLAB X IDE, the output window shows the options passed to the assembler.

3.2 Assembler Interface Syntax

The assembler command line may contain options and the names of source files and has the general form:

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

The order of the assembly source files determines the order in which they are assembled. The special character sequence, `--` (two hyphens not followed by any non-space character) represents the standard input file, and can be used to indicate that source code will be passed to the assembler via the `stdin` stream.

Except for `--` (as described above), any command line argument that begins with a hyphen (–) is an option. The options are case sensitive. It is customary to declare options before the source files; however, this is not mandatory. Each option changes the behavior of the assembler, but no option changes the way another option works.

Some single-letter options require exactly one file name argument to follow them. The file name may either immediately follow the option or it may be separated by a space character. For example, to specify an output file named `test.o`, either of the following options would be acceptable:

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

3.3 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 the `xc32-gcc` driver, which determines from an input file's `.S` or `.s` extension that the file contain assembly code that should be assembled. 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 file name extensions are case sensitive.

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

```
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 compiler user's guide relevant for your target device.

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

3.4 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.

3.4.1 A: Listing Option

The `-a[`*`suboption`*`] [=`*`file`*`]` option enables an assembler list file output and controls what is included in the listing. The forms of this option and their meaning are given in the following table.

`-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`.

3.4.1.1 Omit False Conditionals Listing Option

The `-ac` form of this option 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. The following example shows a listing where the `-ac` option was not used.

```
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
```

The next example shows a listing for the same source where the -ac option was used.

```
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.

3.4.1.2 Omit Debugging Directives Listing Option

The -ad form of this option omits debugging directives from the listing. This option is useful when processing compiler-generated assembly code containing debugging information, as the compiler-generated debugging directives will not clutter the listing. The following example 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.

```
GAS LISTING test.s     page 1
1                  .section .mdebug.abi32
2                  .previous
10                  .Ltext0:
11                  .align    2
12                  .globl    main
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                  .set    nomips16
17                  .ent    main
18                  main:
19         .frame    $fp,8,$31        # vars= 0, regs= 1/0, args= 0, gp= 0
20         .mask    0x40000000,-8
21         .fmask    0x00000000,0
22                  .set    noreorder
23                  .set    nomacro
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                  .set    macro
45                  .set    reorder
46                  .end    main
47                  .LFE0:
49
50                  .comm    testval,4,4
88                  .Letext0:
```

3.4.1.3 High-level language Listing Option

The `-ah` form of this option 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. The following example shows a listing that was generated using the `-alh` command line option.

Example 3-1. 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                   .align    2
12                   .globl    main
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                   .set    nomips16
17                   .ent    main
18                  main:
19         .frame    $sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0
20                  .mask    0x00000000,0
21                   .fmask    0x00000000,0
22                   .set    noreorder
23                   .set    nomacro
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                  .set    macro
37                  .set    reorder
38                  .end    main
39                  .LFE0:
40                  .size    main, .-main
41
42                  .comm    testval,4,4
```

3.4.1.4 Assembly Listing Option

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

3.4.1.5 Expand Macros Listing Option

The `-am` form of this option expands macros in a listing. The first example below shows a listing where the `-am` option was not used.

```
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
```

This second example shows a listing for the same source where the `-am` option was used.

```
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: The > signifies expanded macro instructions.

3.4.1.6 Disable Directive Listing Option

The `-an` form of this option turns off all forms processing that would be performed by the listing directives `.psize`, `.eject`, `.title` and `.sbttl`. The first example below shows a listing where the `-an` option was not used.

```
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
```

This second example shows a listing for the same source where the `-aln` option was used.

```
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
```

3.4.1.7 Symbol Table Listing Option

The `-as` form of this option requests a symbol table listing. The following example shows a listing that was generated using the `-as` command line option. Note that both defined and undefined symbols are listed.

```
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
```

3.4.1.8 Specify Output Listing Option

The `-a=`*`file`* form of this option defines the name of an output file that will be used to store the listing. This file must be in the current directory.

3.4.2 Listing-lhs-width Option

The `--listing-lhs-width=`*`num`* option is used to set the maximum word width of the first output data column in the assembler list file, where *`num`* is the number of words. 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
```

3.4.3 Listing-lhs-width2 Option

The `--listing-lhs-width2=`*`num`* option is used to set the maximum width of any further lines of the hex byte dump for a given input source line in the list file, where *`num`* is the number of words. If this value is not specified, it defaults to being the same as the value specified for `--listing-lhs-width`, or the value 1 if neither option is used. 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
```

3.4.4 Listing-rhs-width Option

The `--listing-rhs-width=`*`num`* 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.

3.4.5 Listing-cont-lines Option

The `--listing-cont-lines=`*num* 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
```

3.5 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`.

3.5.1 Fatal-warnings Option

The `--fatal-warnings` option specifies that the assembler treat warnings as if they were errors. This will cause the assembler to exit without generating a final output.

3.5.2 Warn Option

The `--warn` option requests that all warnings be issued by the assembler. This is the default behavior if no option is specified.

The `--no-warn` form of this option (or its alias `-W`) suppresses the reporting of any warnings. Errors are still reported, and this option does not affect how your source code is assembled. Use this form of the option with care, as some warning messages indicate code that might fail during execution.

3.5.3 J: No Warnings on Signed Overflow Option

The `-J` option suppresses warnings regarding signed overflow.

3.5.4 Help Option

The `--help` option will show the command line usage and options for the assembler, then exit.

3.5.5 Target-help Option

The `--target-help` option when used with no option to define the target processor will show information regarding the PIC32M target-specific command-line options for the assembler, then exit. If the `-mprocessor` option is used to specify a PIC32C/SAM device, the option help displayed will be specific to the ARM family of devices.

3.5.6 Version Option

The `--version` option (or its alias `-v`) prints information pertaining to the assembler's version, then exits.

3.5.7 Verbose Option

The `--verbose` option prints additional information during the assembly process. If this is the only command line option used, then the assembler will print out a banner shower the assembler's version and then wait for entry of the assembly source through the standard input stream. Use `<CTRL>-D`to send an EOF character to end the assembly input.

3.6 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.

3.6.1 G: Generate Debug Information Option

The `-g` option generates symbolic debugging information.

3.6.2 Keep-locals Option

The `--keep-locals` option (or its alias `-L`) prevents local symbols from being discarded from the object files. Such labels begin with `.L` (upper case only). Normally, these labels are not seen when debugging, as they are intended for the use of programs (like compilers) that compose assembler programs.

3.6.3 O: Specify Output Name Option

The `-o` *`objfile`* option renames the object file output to be *`objfile`* instead of `a.out`, which is the default name if no option is specified. Whatever the object file is called, the assembler overwrites any existing file with the same name.

3.6.4 Z: Produce Output on Errors Option

The `-Z` option forces the generation of an output object file, even if errors were encountered during the build. After an error message, the assembler normally produces no output. If any errors are encountered when using this option, the assembler writes an object file after a final warning message of the form, *`n`* `errors,` *`m`* `warnings, generating bad object file`.

3.6.5 MD: Create Dependency File

The `-MD` *`file`* option writes dependency information to *`file`*. A dependency file consists of a single rule suitable for describing the dependencies of the main source file. This feature can be used in the automatic updating of make files.

3.7 Assembler Symbol-Definition and Search-Path Options

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

3.7.1 Defsym Option

The `--defsym` *`sym=value`* option defines the symbol *`sym`* and assigns it *`value`*.

3.7.2 I: Specify Assembler Search Path Option

The `-I` *`dir`* option adds *`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 number 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.

3.8 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 compiler user's guide relevant for your target device for more information on the compilation driver and its options.

3.8.1 Processor Option

The `-mprocessor=device` option selects the target device for which to compile. A list of all supported devices can be found the compiler release notes. Note that the name must be in upper case, for example, `-mprocessor=ATSAME70N20B`.

3.8.2 Wa: Pass Option To The Assembler, Option

The `-Wa,option` option passes its `option` argument directly to the assembler. If `option` contains commas, it is split into multiple options at the commas. For example `-Wa,-a` will pass the `-a` option to the assembler, requesting that an assembly list file be produced.

3.8.3 D: Define a Macro

The `-Dmacro` option allows you to define a preprocessor macro and the `-Dmacro=text` form of this option additionally allows a user-define replacement string to be specified with the macro. A space may be present between the option and macro name.

When no replacement text follows the macro name, the `-Dmacro` option defines a preprocessor macro called `macro` and specifies its replacement text as `1`. Its use is the equivalent of placing `#define macro 1` at the top of each module being compiled.

The `-Dmacro=text` form of this option defines a preprocessor macro called `macro` with the replacement text specified. Its use is the equivalent of placing `#define macro text` at the top of each module being compiled.

Either form of this option creates an identifier (the macro name) whose definition can be checked by `#ifdef` or `#ifndef` directives. For example, when using the option, `-DMY_MACRO` (or `-D MY_MACRO`) and building the following code:

```
#ifdef MY_MACRO
int input = MY_MACRO;
#endif
```

the definition of the `int` variable `input` will be compiled, and the variable assigned the value `1`.

If the above example code was instead compiled with the option `-DMY_MACRO=0x100`, then the variable definition that would ultimately be compiled would be: `int input = 0x100;`

See 4.1.1. Preprocessor Arithmetic for clarification of how the replacement text might be used.

Defining macros as C string literals requires escaping the quote characters (`" "`) used by the string. If a quote is intended to be included and passed to the compiler, it should be escaped with a backslash character (`\`). If a string includes a space character, the string should have additional quotes around it.

For example, to pass the C string, `"hello world"`, (including the quote characters and the space in the replacement text), use `-DMY_STRING="\"hello world\""`. You could also place quotes around the entire option: `"-DMY_STRING=\"hello world\""`. These formats can be used on any platform. Escaping the space character, as in `-DMY_STRING=\"hello\ world\"` is only permitted with macOS and Linux systems and will not work under Windows, and hence it is recommended that the entire option be quoted to ensure portability.

All instances of -D on the command line are processed before any -U options.

3.8.4 U: Undefine Macros

The -U*macro* option undefines the macro macro.

3.8.5 I: Specify Include File Search Path Option

The -I*dir* option adds the directory *dir* to the head of the list of directories to be searched for header files. A space may be present between the option and directory name.

The option can specify either an absolute or relative path and it can be used more than once if multiple additional directories are to be searched, in which case they are scanned from left to right. The standard system directories are searched after scanning the directories specified with this option.

Under the Windows OS, the use of the directory backslash character may unintentionally form an escape sequence. To specify an include file path that ends with a directory separator character and which is quoted, use -I "E:\\", for example, instead of -I "E:\", to avoid the escape sequence \“. Note that MPLAB X IDE will quote any include file path you specify in the project properties and that search paths are relative to the output directory, not the project directory.

Note: Do not use this option to specify any MPLAB XC32 system include paths. The compiler drivers automatically select the default language libraries and their respective include-file directory for you. Manually adding a system include path may disrupt this mechanism and cause the incorrect files to be compiled into your project, causing a conflict between the include files and the library. Note that adding a system include path to your project properties has never been a recommended practice.

3.8.6 Save-temps Option

The -save-temps option instructs the compiler to keep temporary files after compilation has finished. You might find the generated .i and .s temporary files in particular useful for troubleshooting, and they are often used by the Microchip Support team when you enter a support ticket.

The intermediate files will be placed in the current directory and have a name based on the corresponding source file. Thus, compiling foo.c with -save-temps would produce foo.i, foo.s and the foo.o object file.

The -save-temps=cwd option is equivalent to -save-temps.

3.8.7 V: Verbose Compilation

The -v option specifies verbose compilation.

When this option is used, the name and path of the internal linker applications will be displayed as they are executed, followed by the command-line arguments that each application was passed.

You might use this option to confirm that your driver options have been processed as you expect, or to see which internal application is issuing a warning or error.

3.8.8 Help

The --help option displays information on the xc32-gcc linker options, then the driver will terminate.

For example:

```
xc32-gcc --help
Microchip Language Tool Shell Version 4.20 (Build date: Sep 16 2022).
Copyright (c) 2012-2017 Microchip Technology Inc. All rights reserved

  -omf=elf        Select elf object module format
Usage: pic32m-gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
  --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...].
                           Display specific types of command line options.
  (Use '-v --help' to display command line options of sub-processes).
  --version                Display compiler version information.
  -dumpspecs               Display all of the built in spec strings.
  -dumpversion             Display the version of the compiler.
  -dumpmachine             Display the compiler's target processor.
  -print-search-dirs       Display the directories in the compiler's search path.
  -print-libgcc-file-name  Display the name of the compiler's companion library.
```

4. MPLAB XC32 Assembly Language

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.

4.1 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 5. Assembler Directives.

You can use the C compiler driver to perform other C-style preprocessing by ensuring the input file uses a `.S` extension. For additional information on the compilation driver, see the compiler user's guide relevant for your target device.

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.

4.1.1 Preprocessor Arithmetic

Preprocessor macro replacement expressions are textual and do not utilize types. Unless they are part of the controlling expression to the inclusion directives (discussed below), macros are not evaluated by the preprocessor. Once macros have been textually expanded and preprocessing is complete, the expansion forms a C expression which is evaluated by the code generator along with other C code. Tokens within the expanded C expression inherit a type, with values then subject to integral promotion and type conversion in the usual way.

If a macro is part of the controlling expression to a conditional inclusion directive (`#if` or `#elif`), the macro must be evaluated by the preprocessor. The result of this evaluation is often different to the C-domain result for the same sequence. The preprocessor assigns sizes to literal values in the controlling expression that are equal to the largest integer size accepted by the compiler, as specified by the size of `intmax_t` defined in `<stdint.h>`.

4.2 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]
```

4.2.1 Labels

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 4.2.1.1. Local Labels 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.

4.2.1.1 Local Labels

Local labels are used when a temporary scope for a label is needed.

To define a local label, use a numerical symbol as the label name, for example `N:`, (where `N` is a number).

To refer to the most recent previous definition of the label `N:`, use `Nb`. The `b` stands for "backwards." To refer to the next definition of a local label, write `Nf` ("forwards").

There is no restriction on how you use and reuse labels. You can repeatedly define the same local label (using the same number `N`), although you must either refer to 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 more efficiently manner than the others.

Here is an example of local labels being defined and jumps to use the branch instruction, `b`:

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

Which is the equivalent code to the following, which used ordinary lables:

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

```
label_3:
  b label_4
label_4:
  b label_3
```

Local label 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.

4.2.2 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 device assembly language. Mnemonics are not case sensitive.

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

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]
```

4.2.3 Pseudo Instructions

The assembler synthesizes PIC32M pseudo instructions as shown in the following table.

`li rt, imm`	32-bit load immediate
`move rt, rs`	copies rs into `rt`
`la rt, addr`	a load from memory location
`jrs`	jump register jr `rs`
`abs rt, rs`	absolute value, `rt` = |`rs`|
`beq{l} rs, label`	branch equal to zero if `rs` == 0
`bge{ul} rs, rt, label`	branch greater than equal if `rs` >= `rt`
`bgt{ul} rs, rt, label`	branch greater than if `rs` > `rt`
`ble{ul} rs, rt, label`	branch less than equal if `rs` <= `rt`
`blt{ul} rs, rt, label`	branch less than if `rs` < `rt`
`bne{ul} rs, rt, label`	branch not equal to if `rs` != `rt`
`neg{u} rt`	negate rs place in rs `rs` = -`rs`
`rotr/ror rt, rs, n`	rotate right, rotate `rs` *n* bits right, result in `rt` #only pseudo insn before mips32r2
`rotl/rol rt, rs, n`	rotate left, rotate `rs` *n* bits left, result in `rt` #only pseudo insn before mips32r2
`sge{u} rd, rs, rt`	set greater than equal sets `rd` = 1 if `rs` >= `rt`
`sgt{u} rd, rs, rt`	set greater than sets `rd` = 1 if `rs` > `rt`
`sle{u} rd, rs, rt`	set less than equal sets `rd` = 1 if `rs` <= `rt`
`sne{u} rd, rs, rt`	set not equal sets `rd` = 1 if `rs` != `rt`

In the above, the bracketed letter `u` indicates unsigned operation and the letter `l` inidcates that the likely delay slot is only executed if the branch is taken.

Additionally, a two-operand form of all math and logic instructions is permitted, where the first register represents both rd and rt in the full 3-operand form of the instruction. For example `add $t0, $t1` is encoded as `add $t0, $t0, $t1`.

Pseudo instructions for PIC32C/SAM devices are listed below.

`adr{cond}{.W} Rd, label`	Register version - Load address into *Rd*
`adr{cond} Rd, label`	PC relative version - Load Address into *Rd*
`adrl{cond} Rd, label`	Long version - Generate two instructions to load larger addresses into *Rd*
`cpy{cond} Rd, Rm`	Alias for `mov` instruction
`ldr{cond}{.W} Rt, =expr`	Loads Rt with 32-bit immediate or an address
`ldr{cond}{.W} Rt, =label_expr`	If `mov` or `mvn` cannot be used will use literal pool to place value and load from there
`mov32{cond} Rd, expr`	Load any 32-bit immediate or 32-bit address into *Rd*
`neg{cond} Rd, Rm`	*Rd*= -*Rm*, alias for `RSBS{cond} Rd, Rm, #0`
`und{cond}{.W} {#expr}`	Generate an undefined instruction for target arch - useful for forcing hard faults *expr* is used to set condition codes

In the above, the `{cond}` field is comprised of instruction suffices that control conditional execution and condition flag updates. This can be any of the following.

`S`	Set the condition flags after execution
`S{flag}`	Conditionally execute the instruction based on the *flag* condition code suffix, update the condition flags after execution
`{flag}`	Conditionally execute the instruction based on the *flag* condition code suffix , do not update the condition flags after execution
`{.W}`	Specify width of the generated instruction to be 16 or 32 bit

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 5.12. Symbols that Control Code Generation.

4.2.4 Directives

Assembler directives are assembler 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 5. Assembler Directives on the available directives.

4.2.5 Operands

Each machine instruction takes from 0 up to 4 operands. (See the appropriate data sheet of your target device 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 instructions, an operand consists of a core general-purpose register, label, literal, or basereg+offset.

4.2.5.1 General-Purpose Register Operands

PIC32M Registers

Register operands can be specified using the appropriate register number preceded by a dollar sign (`$`).The following example shows assembly source code using register number operands.

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

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. These macros map conventional register names to the corresponding register number. The following example shows assembly source code using conventional register names for operands. See the *MPLAB® XC32 C/C++ Compiler User's Guide for PIC32/SAM MCUs* for additional information on PIC32M register conventions and the compiler's runtime environment.

```
#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 */
```

Note that these macro names cannot ordinarily be used within assembly code placed inline with C code using the `__asm()` statement.

PIC32C/SAM Registers

Register operands are specified using the appropriate register number preceded by a `r` character. For example the following shows assembly code using several register operands.

```
  ldr    r2, [r7, #4]
  ldr    r3, [r7]
  add    r3, r3, r2
```

Special symbols can be used to represent generic `rx` registers that have a dedicated purpose. These are listed in the following table.

Special symbol	Generic symbol	Usage
sp	r13	stack pointer
lr	r14	link register
pc	r15	program counter

4.2.5.2 Literal-Value Operands

For PIC32M devices, literal operands consist of just the literal value. For example:

```
  .align  2      # literal used with directive
  addiu   $sp,$sp,-8      # literal used with instruction
```

For PIC32C/SAM devices, literal operands for device instructions consists of the `#` character followed by the literal value. In directives, the operand consists of just the literal value.

```
  .align  1      @ literal used with directive
  sub     sp,sp,#12       @ literal used with instruction
```

In all cases, the value can be entered with any of the formats described in 4.3. Constants, provided that format is appropriate for the instruction or directive in which it is being used.

4.2.5.3 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 PIC32M data sheet shows this type of operand as `Mem[R+offset]`.

The following example shows 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-relative) 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 `-G` option lets you change the maximum size of global and static data items that can be addressed in one instruction instead of two from the default value of 8 bytes, which is large enough to hold all simple scalar variables.

The following example shows 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
```

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*for more information on the global pointer and the `-G` option.

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 not more than the number of bytes specified by the `-G` option 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.

4.2.6 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 5. Assembler Directives.

4.2.7 Comments

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

4.2.7.1 Single-line Comments

A single-line comment extends from the comment character to the end of the line. For PIC32M devices, the comment character is a number/hash sign (`#`), for example:

```
addiu  $2,$2,7   # offset the value by threshold
```

For PIC32C/SAM devices, it is an 'at' symbol (`@`), for example:

```
mov    r0, r3    @ put the result in the correct register
```

4.2.7.2 Multi-line Comment

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

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

For example:

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

4.3 Constants

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

4.3.1 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.

4.3.1.1 Integers

The size and format of integer numbers specified in assembler source is the same as that specified by a `long` type in the MPLAB XC32 implementation of the C language. Integer constants can be specified in one of several radices.

A binary integer is specified as either prefix `0b` or `0B` followed by zero or more binary digits (`0` or `1`).

An octal integer is specified as the prefix 0 followed by zero or more octal digits (0 - 7).

A decimal integer must start with a non-zero decimal digit (1 - 9) followed by zero or more decimal digits (0 - 9).

A hexadecimal integer is specified as either prefix 0x or 0X followed by one or more hexadecimal digits (0 - 9, a - f, or A - F).

To denote a negative integer, use the prefix operator –.

4.3.1.2 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 (0 - 9).
- 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.

4.3.2 Character Constants

There are two kinds of character constants. A *character* represents an integer value in a character set that may be used in numeric expressions. A *string* consists of a sequence of characters can contain potentially many bytes and their values may not be used in arithmetic expressions.

4.3.2.1 Characters

A character constant can be written as the character preceded by a single quote character, for example, 'z, or as the character enclosed by single quote characters, for example, 'z'.

Special control characters an be specified by using an escape sequence. The assembler accepts the following escape characters.

Table 4-1. 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

...........continued

Escape Character	Description	Hex Value
`\"`	Double quote character	22
`\`*`digit digit digit`*	Octal character code. The numeric code is 3 octal digits (`0` - `7`).	
`\x` *`hex-digits`* or `\X` *`hex-digits`*	Hex character code. All trailing hexadecimal digits (`0` - `9`, `a` - `f`, or `A` - `F`)are combined.	

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 set is ASCII.

4.3.2.2 Strings

A string is a sequence of characters, not including carriage return or newline characters, enclosed within matching quotes. It may contain double quotes or null characters. To include special characters into a string, escape the characters with a backslash '\' character, as described in 4.3.2.1. Characters.

4.3.2.3 General Syntax Rules

The following table summarizes the general syntax rules that apply to the assembler:

Table 4-2. Syntax Rules

Character	Character Description	Syntax Usage
`.`	period	begins a directive
`#`	number/hash	begin single-line comment
`//` or `@`	Double slash or at symbol	Begin single-line comments in PIC32C/SAM source
`/*`	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

4.4 Symbols

A symbol represents a value and can be comprised of 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 the letters is significant (for example, `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.

4.4.1 Assigning Values to Symbols

A symbol can be given an arbitrary value by assigning the result of an expression to it using the equals sign `=`, for example:

```
VAR = 4
```

4.4.2 The Special Dot Symbol

The special symbol `.` refers to the current address being processed by the assembler. Thus, the expression `melvin: .long .` defines a label `melvin` and places the address of that label in the output. 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. Ensure that any instructions following where the dot symbol has been modified are properly aligned to the instruction width. For example, if a section contains instructions that are 32-bit wide, the address should be aligned to 4 bytes before any instructions are placed, for example:

```
.=.+20
.align 4
moreCode:
  li   $2, 55
```

4.4.3 Predefined Symbols

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

The following table lists symbols usable with PIC32M devices.

Table 4-3. 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

4.5 Expressions

An expression consists of operators and numeric or symbolic operands, and which evaluates to 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.

4.5.1 Empty Expressions

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

4.5.2 Integer Expressions

An integer expression is one or more arguments delimited by operators, yielding an integer value. 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 bytes for each instruction word, but Thumb instructions on SAM devices are typically 2 bytes long. For example, to branch to the next 32-bit instruction after label `L`, specify `L+4` as the destination, for example:

```
b   L+4
```

4.6 Operators

Operators are arithmetic functions, like + or %. Prefix, or unary, operators are followed by a single operand. Infix, or binary, operators appear between their two operands. Operators may be preceded and/or followed by white space.

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

4.6.1 Prefix Operators

Prefix operators are unary operators and operate on one absolute operand, which follows the operator in the source code. The assembler has the following prefix operators.

Table 4-4. Prefix Operators

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

4.6.2 Infix Operators

Infix operators are binary operators and operate on two operands, 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 as they appear in the source. Apart from with the + or - operators, both operands must be absolute, and the result is absolute.

Table 4-5. Infix Operators

Operator	Description	Example
Arithmetic Operators		
*	Multiplication	5 * 4 (=20)
/	Division. Truncation is the same as the C operator '/'.	23 / 4 (=5)
%	Remainder	30 % 4 (=2)
<<	Shift Left. Same as the C operator '<<'	2 << 1 (=4)
>>	Shift Right. Same as the C operator '>>'	2 >> 1 (=1)
Bitwise Operators		
&	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 Operators		
+	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 Operators		
==	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)

...........continued

Operator	Description	Example
Logical Operators		
`&&`	Logical AND	`.if ((x > 1) && (x < 10))`
`||`	Logical OR	`.if ((y != x) || (y < 100))`

4.7 Special Operators

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

Table 4-6. Special Operators (Device Dependent)

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`*

4.7.1 Sizeof Operator

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

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

4.7.2 Startof Operator

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

```
.word .startof.(.data)
```

4.7.3 Endof Operator

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

```
.word .endof.(.data)
```

5. Assembler Directives

Assembler directives are tool commands, which appear in the source code but which are not usually translated directly into instruction opcodes. They are used to control the assembler's input, output, and data allocation.

All assembler directives have names that begin with a period (`.`). The names are case insensitive but are usually written in lower case.

Note: Assembler directives are ***not*** target instructions (such as `ADD`, `XOR`, `JAL`, etc.). For instruction set information, consult the data sheet for your target device.

5.1 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.

The section directives follow.

5.1.1 Bss Directive

The `.bss` directive indicates that the output of the assembly code following should be placed at 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, the entire `.bss` section is filled with zeroes.

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 either `.skip` or `.space` directives

```
# 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
```

5.1.2 Data Directive

The `.data` indicates that the output of the assembly code following should be placed at the end of the `.data` (initialized data) section.

```
# 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 10.4.3. Run-Time Library Support.

5.1.3 Pushsection Directive

The `.pushsection` *`name`* 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`.

5.1.4 Popsection Directive

The `.popsection` directive replaces the current section description with the section stored on the top of the section stack. This section is then popped off the stack.

5.1.5 Section Directive

The `.section name [, attr`$_1$`[,...,attr`$_n$`]]` directive, where the optional argument is not quoted, assembles the code following a section with *`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.scn`*`n`*, where *`n`* represents the number of auto-
generated section names.

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

If the optional argument is not present, the section attributes depend on the section name. A table of reserved section names with implied attributes is given in 9.3. Default Linker Script. 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 16. 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.

5.1.5.1 Section Type Attributes

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

Table 5-1. Attributes the 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

5.1.5.2 Section Type Modifier Attributes

Attributes can be used to modify some sections, as indicated in the following table.

Table 5-2. Attributes that Modify Section Types

Attribute	Description	Applicable Sections				
		`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	

...........continued

Attribute	Description	Applicable Sections				
		code	data	bss	persist	ramfunc
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

Attributes may be combined with others in some cases. The following table indicates which attributes are compatible with each other.

Table 5-3. 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	

5.1.5.3 Section Directive Examples

The following show examples of the `.section` directive being used to specify sections for output.

```
.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
```

5.1.6 Text Directive

Definition

The `.text` directive indicates that the output of the assembly code following should be placed at the end of the `.text` (executable code) section, for example.

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

5.2 Directives that Initialize Constants

5.2.1 Ascii Directive

The `.ascii "`*string*$_1$`" [, ..., "`*string*$_n$`"]` directive assembles each comma-separated string argument into consecutive addresses in the current section. A trailing zero byte is not automatically added.

5.2.2 Asciz Directive

The `.asciz "`*string*$_1$`" [, ..., "`*string*$_n$`"]` directive produces similar output to the `.ascii` directive, but each string is followed by a zero byte. The “`z`” in `.asciz` stands for “zero.” This directive is a synonym for `.string`.

5.2.3 Byte Directive

The `.byte` *expr*$_1$`[, ...,` *expr*$_n$`]` directive evaluates the specified comma-separated expressions and stores these as bytes into the current section.

5.2.4 Double Directive

The `.double` *value*$_1$`[, ...,` *value*$_n$`]` directive assembles the specified comma-separated double-precision (64-bit) floating-point constants into consecutive addresses in the current section. Floating point numbers are stored in IEEE format (see 4.3.1.2. Floating-Point Numbers) as little-endian values.

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
```

5.2.5 Float Directive

The `.float` *value*$_1$`[, ...,` *value*$_n$`]` directive assembles the specified comma-separated single-precision (32-bit) floating-point constants into consecutive addresses in the current section. Floating point numbers are stored in IEEE format (see 4.3.1.2. Floating-Point Numbers) as little-endian values. This directive has the same effect as `.single`.

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
```

5.2.6 Single Directive

The `.single` *value*$_1$`[, ...,` *value*$_n$`]` directive the specified comma-separated single-precision (32-bit) floating-point constants into consecutive addresses in the current section. This directive is a synonym for `.float` (see 5.2.5. Float Directive).

5.2.7 Hword Directive

The `.hword` *expr*$_1$`[, ...,` *expr*$_n$`]` directive evaluates and assembles one or more expressions into 2-byte numbers, stored into consecutive addresses in little-endian format. This directive is a synonym for `.short`.

5.2.8 Int Directive

The `.int` *expr*$_1$`[, ...,` *expr*$_n$`]` directive evaluates and assembles one or more expressions into 4-byte numbers, stored in consecutive addresses in little-endian format. This directive is a synonym for `.long`.

5.2.9 Long Directive

The `.long` *expr*$_1$`[, ...,` *expr*$_n$`]` directive evaluates and assembles one or more expressions into 4-byte numbers, stored in consecutive addresses in little-endian format. This directive is a synonym for `.int`.

5.2.10 Short Directive

The `.short` *expr*$_1$`[, ...,` *expr*$_n$`]` directive evaluates and assembles one or more expressions into 2-byte numbers, stored into consecutive addresses in little-endian format. This directive is a synonym for `.hword`.

5.2.11 String Directive

The `.string "`*str*`"[, "`*str*`"]` directive assembles each comma-separated string argument into consecutive addresses in the current section. A zero byte will be added to each string. This directive is a synonym for the `.asciz` directive.

5.2.12 Word Directive

The `.word` *expr*$_1$`[, ...,` *expr*$_n$`]` directive evaluates and assembles one or more expressions into 4-byte numbers, stored into consecutive addresses in little-endian format.

5.3 Directives that Declare Symbols

5.3.1 Comm Directive

The `.comm` *symbol*`,` *length* `[,` *algn*`]` directive 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.

5.3.2 Extern Directive

The `.extern` *symbol* directive declares a symbol name that can be used in the current module, but it is defined as global in a different module. All symbols are `extern` by default so use of this directive is optional.

5.3.3 Global/Globl Directives

The `.global` *symbol*
(and its alias, `.globl` *symbol*) 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.

5.3.4 Lcomm Directive

The `.lcomm` *`symbol`*`,` *`length`* directive reserves *`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.

5.3.5 Weak Directive

The `.weak` *`symbol`* directive marks the symbol named *`symbol`* as weak. When a weak-defined symbol is used and linked with a definition for that symbol, that symbol definition is used with no error. When a weak-defined symbol is used and the symbol is not defined, the value of the weak symbol becomes zero with no error.

5.4 Directives that Define Symbols

5.4.1 Equ Directive

The `.equ` *`symbol`*`,` *`expression`* directive sets the value of *`symbol`* to *`expression`*. You may set the same 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. This directive is equivalent to the `.set` directive (see 5.4.3. Set Directive).

5.4.2 Equiv Directive

The `.equiv` *`symbol`*`,` *`expression`* directive operates like the `.equ` directive, 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
```

5.4.3 Set Directive

The `.set` *`symbol`*`,` *`expression`* directive sets the value of *`symbol`* to *`expression`*. You may set the same 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. This directive is equivalent to the `.equ` directive (see 5.4.1. Equ Directive).

5.5 Directives that Modify Section Alignment

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

5.5.1 Align Directive

The `.align [`*`algn`*`[,` *`fill`*`]]` 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` directive.

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. You may wish to use a fill value of 0xFF for regions of Flash memory.

5.5.2 Fill Directive

The `.fill repeat[, size[, value]]` directive reserves space for `repeat` copies of a `size`-byte value derived from `value`. The repeated value is the lowest `size` bytes of a number sequence consisting of the highest order 4 bytes being zero and the lowest order 4 bytes being `value` rendered in little-endian byte-order.

The `size` and `value` arguments are optional. When not specified, the `size` defaults to the value 1 and the `value` argument defaults to value 0.

The `repeat` argument is mandatory and may be zero or more, but if it is more than 8, then it is deemed to have the value 8.

For example, the following code will place down 3 copies of the byte 0xFF.

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

5.5.3 Org Directive

The `.org new-lc[, fill]` directive advances the location counter of the current section to `new-lc`. The `new-lc` argument 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.

5.5.4 Skip Directive

The `.skip size[, fill]` directive emits `size` bytes, each of value `fill`. This directive is identical to the .space directive (see 5.5.5. Space Directive).

5.5.5 Space Directive

The `.space size[, fill]` directive emits `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.

5.5.6 Struct Directive

The `.struct expression` directive switches to the absolute section and sets 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 code 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.

5.6 Directives that Format the Output Listing

5.6.1 Eject Directive

The `.eject` directive forces a page break at this point when generating assembly listings.

5.6.2 List Directive

The `.list` directive 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.

This directive is 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.

5.6.3 Nolist Directive

The `.nolist` directive 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.

This directive is 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.

5.6.4 Psize Directive

The `.psize` *`lines`*`[,` *`columns`*`]` directive 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 form feeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

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

5.6.5 Sbttl Directive

The `.sbttl "`*`subheading`*`"` directive specifies *`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.

5.6.6 Title Directive

The `.title "`*`heading`*`"` directive specifies *`heading`* as the title (second line, immediately after the source file name and page number) when generating assembly listings.

5.7 Directives that Control Conditional Assembly

5.7.1 Else Directive

The `.else` directive is used in conjunction with the `.if` directive to provide an alternative path of assembly code should the `.if` evaluate to false.

5.7.2 Elseif Directive

The `.elseif` *`expr`* directive is 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.

5.7.3 Endif Directive

The `.endif` directive marks the end of a block of code that is conditionally assembled.

5.7.4 If Directive

The `.if` *`expr`* directive 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` or `.elseif`.

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

5.7.4.1 Ifdef Directive

The `.ifdef` *`symbol`* directive assembles the following section of code up until the next `.endif` directive if the specified symbol has been defined. Note that a symbol that has been referenced but not yet defined is considered to be undefined.

5.7.4.2 Ifc Directive

The `.ifc` *`string1`*`,`*`string2`* directive assembles the following section of code if the two strings are identical. 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 white space should be quoted. The string comparison is case sensitive.

5.7.4.3 Ifeq Directive

The `.ifeq` *`expr`* directive assembles the section of code following if the argument expression evaluates zero. The argument must be an absolute expression.

5.7.4.4 Ifeqs Directive

The `.ifeqs "`*`string1`*`","`*`string2`*`"` directive is equivalent to the `.ifc` directive. The strings must be quoted using double quotes.

5.7.4.5 Ifge Directive

The `.ifge` *`expr`* directive assembles the following section of code if the argument is greater than or equal to zero. The argument must be an absolute expression.

5.7.4.6 Ifgt Directive

The `.ifgt` *`expr`* directive assembles the following section of code if the argument is greater than zero. the argument must be an absolute expression.

5.7.4.7 Ifle Directive

The `.ifle` *`expr`* directive assembles the following section of code if the argument is less than or equal to zero. The argument must be an absolute expression.

5.7.4.8 Iflt Directive

The `.iflt` *`expr`* directive assembles the following section of code if the argument is less than zero. The argument must be an absolute expression.

5.7.4.9 Ifnc Directive

The `.ifnc` *`string1`*`,`*`string2`* 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.

5.7.4.10 Ifndef Directive

The `.ifndef` *`symbol`* directive assembles the following section of code if the specified symbol has not been defined. Note a symbol which has been referenced but not yet defined is considered to be undefined.

5.7.4.11 Ifnotdef Directive

The `.ifnotdef` *`symbol`* directive is equivalent to the `.ifndef` directive.

5.7.4.12 Ifne Directive

The `.ifne` *`expr`* directive assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to `.if`). The argument must be an absolute expression.

5.7.4.13 Ifnes Directive

The `.ifnes` *`string1`*`,`*`string2`* directive works similarly to the `.ifeqs` directive, but the sense of the test is reversed. The directive assembles the section of code following if the two strings are not the same.

5.8 Directives for Substitution/Expansion

5.8.1 Exitm Directive

The `.exitm` directive exits early from the current macro definition. See `.macro` directive (5.8.4. Macro Directive).

5.8.2 Irp Directive

The `.irp` *`symbol`*`,` *`value`*`1`
`[, ..., `*`value`*`n]`
`...`
`.endr` directive evaluates 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)
```

5.8.3 Irpc Directive

The `.irpc` *`symbol`*`,` *`value`*
`...`
`.endr` directive 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)
```

5.8.4 Macro Directive

The `.macro` *`name`* `[`*`args`*`]` and `.endm` directives allow you to define macros that generate assembly output.

The macro will be called *`name`*. If required, it can take arguments when it is used. To allow this, follow the macro name with a list of the argument names with either a space character or comma separating them. A default value can be specified with any argument by following its name with `=`*`value`*.

The following are valid opening lines for macro definitions.

- `.macro comm`
  – Begin the definition of a macro called `comm`, which takes no arguments.
- `.macro plus1 p, p1`
  or `.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`).

For example, this definition specifies a macro `SUM` with two arguments, `from` and `to`, that puts a sequence of numbers into memory:

```
.macro SUM from=0, to=3
.long \from
.if    \+o-\from
SUM "(\from+1)",   \+o
.endif
.endm
```

The above macro could be used (called), specifying the arguments values, for example, `SUM 0,5`, which would expand to the following:

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

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=17, from=9`.

The assembler maintains a counter of how many macros it has executed. You can copy that number to the output by using the `\@` operator, 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
```

5.8.5 Endm Directive

The `.endm` directive marks the end of a macro definition (see 5.8.4. Macro Directive).

5.8.6 Endr Directive

The `.endr` directive marks the end of the block of instructions to be repeated using the `.rept` directive (see 5.8.8. Rept Directive).

5.8.7 Purgem Directive

The `.purgem` *`name`* directive undefines the macro *`name`*, so that later uses of the string will not be expanded. See the `.macro` directive (5.8.4. Macro Directive).

5.8.8 Rept Directive

The `.rept` *`count`* directive repeat the sequence of lines between the `.rept` directive and the next `.endr` directive, *`count`* times. Specifying *`count`* as zero is allowed, but no output is generated. Negative values for *`count`* are not permitted and if encountered will be treated as if they were zero.

For example, assembling:

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

is equivalent to assembling:

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

5.9 Directives that Include Other Files

5.9.1 Incbin Directive

The `.incbin "`*`file`*`"[,`*`skip`*`[,`*`count`*`]]` directive includes the content of the file *`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 3. 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.

5.9.2 Include Directive

The `.include "`*`file`*`"` directive provides a way to include a supporting file named *`file`* at a specified point in your source code. The code is assembled with the file's content replacing the `.include` directive. When the end of the included file is reached, assembly of the original file continues at the statement following the `.include`.

5.10 Directives that Control Diagnostic Output

5.10.1 Abort Directive

The `.abort` directive prints out the message `.abort detected. Abandoning ship.`, and exits the program.

5.10.2 Err Directive

The `.err` directive prints an error message. Unless the `-Z` option was used, the error will stop the assembly process and an object file will not be generated. Typically, this directive is used to signal an error in conditionally compiled code.

5.10.3 Error Directive

The `.error "`*`string`*`"` directive is similar to `.err` (see 5.10.2. Err Directive), except that the specified string is printed as the error message.

5.10.4 Fail Directive

The `.fail` *`expression`* directive 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 might be useful inside complex nested macros or conditional assembly.

5.10.5 Ident Directive

The `.ident "`*`comment`*`"` directive 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.

5.10.6 Print Directive

The `.print "`*`string`*`"` directive prints *`string`* on the standard output during assembly.

5.10.7 Version Directive

The `.version "`*`string`*`"` 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.

5.10.8 Warning Directive

The `.warning "`*`string`*`"` directive is similar to the `.error` directive (see 5.10.3. Error Directive), but the string is emitted as a warning.

5.11 Directives for Debug Information

5.11.1 Ent Directive

The `.ent` *`function`* directive is useable with PIC32M devices and marks *`function`* as being a function symbol, in the same way it would with the generic `.type` directive.

5.11.2 End Directive

The `.end` directive indicates the end of the program source code. Nothing beyond this directive will be processed.

5.11.3 File Directive

The `.file` *`fileno`* `"`*`filename`*`"` directive assigns filenames to the `.debug_line` file name table when emitting dwarf2 line-number information. 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.

5.11.4 Fmask Directive

The `.fmask` *`mask`*`,` *`offset`* directive provides MIPS-specific DWARF debugging information in the ELF file, but is not normally used with hand-written assembly source. Maintain a `mask` and `offset` value of 0.

5.11.5 Frame Directive

The `.frame` *`framereg`*`,` *`frameoffset`*`,` *`retreg`* directive provides MIPS-specific DWARF debugging information in the ELF file, but is not normally used with hand-written assembly source. It is useable with PIC32M devices and describes the shape of the stack frame. The virtual frame pointer in specified by *`framereg`* and this will normally be 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`. For example

```
.frame $fp,8,$31
```

5.11.6 Loc Directive

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

5.11.7 Mask Directive

The `.mask` *`mask`*`,` *`offset`* directive provides MIPS-specific DWARF debugging information in the ELF file, but is not normally used with hand-written assembly source. It is useable with PIC32M devices and indicates which of the integer registers are saved in the current function's stack frame. The *`mask`* value is interpreted as a bitmask, with a bit in bit position *`n`* representing register number *`n`*. A bit set to 1 in the bitmask indicates that the corresponding register 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.

5.11.8 Size Directive

The `.size` *`name`*`,` *`expression`* 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.

5.11.9 Sleb128 Directive

The `.sleb128` $expr_1$ `[, ..., ` $expr_n$`]` evaluates the expression arguments as signed little-endian base 128 values and places these in the output. This is a compact, variable-length representation of numbers used by the DWARF symbolic-debugging format.

5.11.10 Type Directive

The `.type name, description` directive sets the type of the symbol `name` to be either a function symbol or an object symbol. There are five different forms of syntax supported for the type `description` field, in order to provide compatibility with various other assemblers, those being:

```
.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
```

5.11.11 Uleb128 Directive

The `.uleb128 expr`$_1$`[,...,expr`$_n$`]` directive evaluates the expression arguments as unsigned little-endian base 128 values and places these in the output. This is a compact, variable-length representation of numbers used by the DWARF symbolic-debugging format.

5.12 Symbols that Control Code Generation

The following sections describe symbols that can be set

5.12.1

The `.set noat` directive prevents the assembler from using

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.

5.12.2 `.set at`

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

5.12.3 `.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.

5.12.4 `.set autoextend`

Enable auto-extension of MIPS16 instructions to 32 bits.

5.12.5 `.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.

5.12.6 `.set macro`

Suppress warnings for synthesized instructions.

5.12.7 `.set mips16e`

Assemble with the MIPS16e ISA extension.

5.12.8 `.set nomips16e`

Do not assemble with the MIPS16e ISA extension.

5.12.9 `.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.

5.12.10 `.set reorder`

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

5.12.11 `.set micromips`

Assemble with the microMIPS ISA mode.

5.12.12 `.set nomicromips`

Do not assemble with the microMIPS ISA mode.

6. Assembler Errors/Warnings/Messages

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.

6.1 Fatal Errors

Fatal errors indicate that an internal error has occurred in the assembler. Please contact Microchip Technology (support.microchip.com) for support if the assembler generates a fatal error, providing full details about the source code and command-line options causing the error.

6.2 Errors

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

6.2.1 Symbol

.abort detected. Abandoning ship.

User error invoked with the `.abort` directive.

.else without matching .if

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

.elseif after .else

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

.elseif without matching .if

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

.endfunc missing for previous .func

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

.endif without .if

An `.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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 D

Duplicate else.

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

6.2.6 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.

6.2.7 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.

6.2.8 G

Global symbols not supported in common sections.

External symbols are not supported in MRI common sections.

6.2.9 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

6.2.10 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.

6.2.11 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.

L**ui expression not in range 0..65535.**

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

6.2.12 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 5.7. 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 5.7. 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.

6.2.13 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.

6.2.14 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.

6.2.15 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 ')'.

6.2.16 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.

6.2.17 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.

6.3 Fatal Errors

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.

6.3.1 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.

6.3.2 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].

6.3.3 D

Divide by zero.

DIV instruction with $zero register as RT.

Division by zero.

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

6.3.4 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.

6.3.5 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.

6.3.6 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.

6.3.7 J

Jump address range overflow (0x%lx).

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

6.3.8 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.

6.3.9 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.

6.3.10 O

Operand overflow.

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

6.3.11 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.

6.3.12 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.

6.3.13 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.

6.3.14 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.

6.4 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.

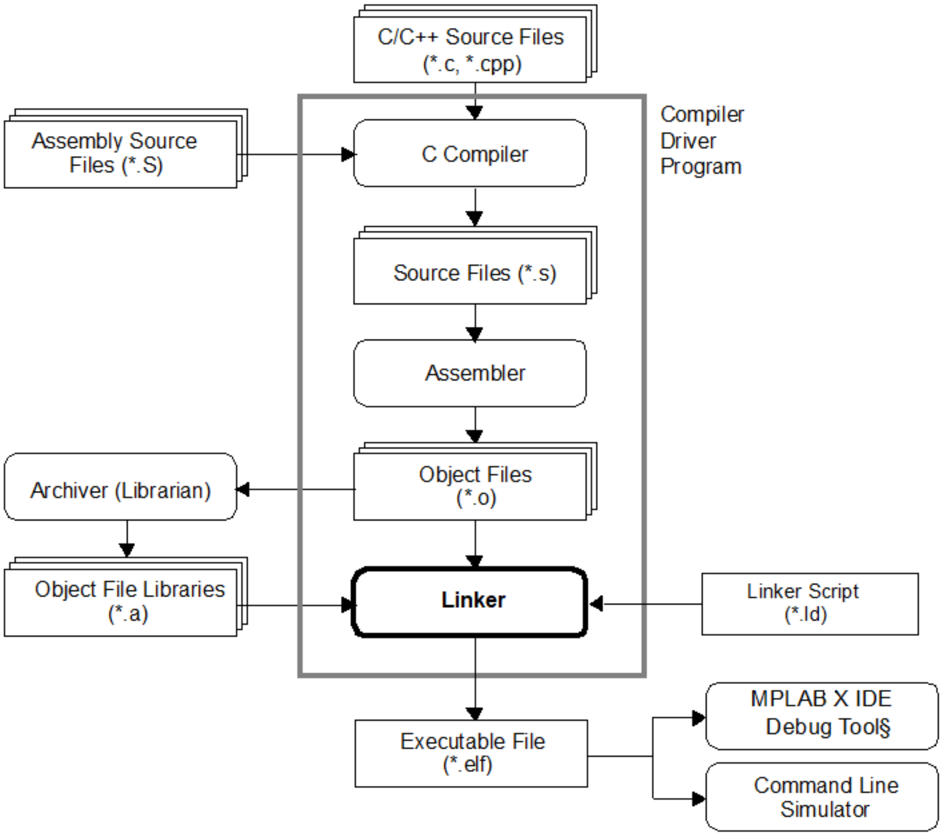
7. Linker Overview

The MPLAB XC32 Object Linker (`xc32-ld`) produces binary code from relocatable object code and archives for the PIC32/SAM MCU family of devices. The linker is a part of the GNU linker from the Free Software Foundation.

7.1 Linker and Other Development Tools

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

Figure 7-1. Linker Tools Process Flow



7.2 Feature Set

Notable features of the linker include:

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

7.3 Input/Output Files

Linker input and output files are listed below.

Table 7-1. Linker Files

Extension	Description
Input Files	
`.o`	Object Files
`.a`	Library Files
`.ld`	Linker Script File (device-specific fragment)
Output Files	
`.elf`, `.out`	Linker Output Files
`.map`	Map File

The 32-bit linker is capable of creating a map file and a binary ELF file (that may or may not contain debugging information). Assembler listings can be generated using the `xc32-objdump` binary utility on either an object or ELF file.

7.3.1 Object Files

Object files contain relocatable (not absolute) binary code/data produced by the assembler. The linker accepts object files in the ELF format.

7.3.2 Library Files

Library (archive) files consist of a collection of relocatable object files grouped together for convenience.

7.3.3 Linker Script File

Linker scripts, or command files, can perform the following tasks:

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

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

Example 7-1. 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
}
```

7.3.4 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.

7.3.5 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-Usage Report by Module Shows the memory usage (text, data, and bss sections) per object file. This report indicates the size in the final ELF file attributable to each input object. A miscellaneous entry also shows those sections whose input object file cannot be determined or the type of memory cannot be established.

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.

A typical map file showing the Memory-Usage Report and the Memory-Usage Report by Module is presented below.

```
Archive member included to satisfy reference by file (symbol)

/Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../pic32c/lib/
thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
                              startup_atsamv71j19b.o (__pic32c_data_initialization)
/Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../pic32c/lib/
thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
```

```
                              /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
(__pic32c_data_initialization_impl)
/Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../pic32c/lib/
thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
                              startup_atsamv71j19b.o (__libc_init_array)
/Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../pic32c/lib/
thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
                              /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
(__progname_full)
/Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../pic32c/lib/
thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)
                              /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o) (__environ)

Microchip PIC32 Memory-Usage Report

ROM Program-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                 ----------  -------------------------  -----------
.vectors                  0x400000          0x168         360
.text                     0x400168           0x20          32  App's exec code
.text                     0x400188          0x110         272  App's exec code
.text.Reset_Handler       0x400298           0x8c         140
.text.__pic32c_data_ini   0x400324           0x68         104
.text.__libc_start_init   0x40038c           0x5c          92
.dinit                    0x4003e8           0x50          80
.text                     0x400438           0x40          64  App's exec code
.text.main                0x400478           0x30          48
.text.__pic32c_data_ini   0x4004a8            0x4           4
.text.Dummy_Handler       0x4004ac            0x2           2
.text.__dummy             0x4004ae            0x2           2
      Total ROM used  :       0x4b0        1200  0.2% of 0x80000
        --------------------------------------------------------------------------
         Total Program Memory used  :       0x4b0        1200  0.2% of 0x80000
        --------------------------------------------------------------------------


RAM Data-Memory Usage
section                    address  length [bytes]      (dec)  Description
-------                 ----------  -------------------------  -----------
.bss                    0x20400000           0x6c         108  Uninitialized data
.bss                    0x2040006c            0x4           4  Uninitialized data
.bss                    0x20400070           0x14          20  Uninitialized data
.bss                    0x20400084            0x4           4  Uninitialized data
         Total RAM used  :        0x88         136  0.1% of 0x40000
        --------------------------------------------------------------------------
            Total Data Memory used  :        0x88         136  0.1% of 0x40000
        --------------------------------------------------------------------------


Dynamic Data-Memory Reservation
section                    address  length [bytes]      (dec)  Description
-------                 ----------  -------------------------  -----------
heap                    0x20400090           0x10          16  Reserved for heap
stack                   0x204000b8        0x3ff40      261952  Reserved for stack

        --------------------------------------------------------------------------

Microchip PIC32 Memory-Usage Report By Module

    text    data     bss     dec     hex    basename                  filename
--------------------------------------------------------------------------------
     774       0       0     774     306    startup_atsamv71j19b.ostartup_atsamv71j19b.o
     104       0       4     108      6c    data_init_0_1.o       libpic32c.a
      72       0      28     100      64    crtbegin.o            /Applications/microchip/
xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
      94       0       0      94      5e    __libc_start_main.o   libc-musl.a
       0       0      80      80      50    libc.o                libc-musl.a
      80       0       0      80      50    data_init             data_init
      48       0      20      68      44    xc32_floats.o         build/default/production/
_ext/1472/xc32_floats.o
      16       0       0      16      10    crtn.o                /Applications/microchip/
xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o
       8       0       0       8       8    crti.o                /Applications/microchip/
xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
       4       0       0       4       4    data_init_wrapper.o   libpic32c.a
       0       0       4       4       4    __environ.o           libc-musl.a
--------------------------------------------------------------------------------
    1200       0     136    1336     538    dist/default/production/
XC32_sam_library_example.X.production.elf


Discarded input sections

 .text          0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
 .data          0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
 .bss           0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
 .xc_stack_usage.hdr
                0x00000000        0xe /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
 .xc_stack_usage
                0x00000000       0x26 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
 .data          0x00000000        0x4 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
 .xc_stack_usage
                0x00000000       0x3c /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
 .xc_stack_usage.hdr
                0x00000000        0xe /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
 .text          0x00000000        0x0 build/default/production/_ext/1472/xc32_floats.o
 .data          0x00000000        0x0 build/default/production/_ext/1472/xc32_floats.o
 .data          0x00000000        0x0 startup_atsamv71j19b.o
 .bss           0x00000000        0x0 startup_atsamv71j19b.o
 .text
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .data
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .bss
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .xc_stack_usage

0x00000000       0x2a /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .xc_stack_usage.hdr

0x00000000        0xe /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .text
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .data
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .xc_stack_usage

0x00000000       0x2f /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .xc_stack_usage.hdr

0x00000000        0xe /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .text
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .data
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .bss
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .xc_stack_usage

0x00000000       0x34 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .xc_stack_usage.hdr
```

```
0x00000000        0xe /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .text
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .data
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .xc_stack_usage.hdr

0x00000000        0xe /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .text
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)
 .data
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)
 .xc_stack_usage.hdr

0x00000000        0xe /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)
 .text          0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .data          0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .bss           0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .eh_frame      0x00000000        0x4 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .xc_stack_usage.hdr
                0x00000000        0xe /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .comment       0x00000000       0x30 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .ARM.attributes
                0x00000000       0x30 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtend.o
 .text          0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o
 .data          0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o
 .bss           0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o
 .xc_stack_usage.hdr
                0x00000000        0xe /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o

Memory Configuration

Name             Origin             Length             Attributes
rom              0x00400000         0x00080000         xrl
ram              0x20400000         0x00040000         xw !r
itcm             0x00000000         0x00000000         xw
dtcm             0x20000000         0x00000000         xw !r
*default*        0x00000000         0xffffffff

Linker script and memory map

LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/
hard/crti.o
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/
hard/crtbegin.o
LOAD build/default/production/_ext/1472/xc32_floats.o
                0x00000001                __MPLAB_BUILD = 0x1
LOAD startup_atsamv71j19b.o
START GROUP
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../
pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../
pic32c/lib/thumb/v7e-m+dp/hard/libm-emfloat.a
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../
pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/../../../../
pic32c/lib/thumb/v7e-m+dp/hard/libm-emfloat.a
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/
hard/libgcc.a
END GROUP
```

```
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/
hard/crtend.o
LOAD /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/
hard/crtn.o
                0x00480000                __rom_end = (ORIGIN (rom) + LENGTH (rom))
                0x20440000                __ram_end = (ORIGIN (ram) + LENGTH (ram))

.vectors        0x00400000      0x168
                0x00400000                . = ALIGN (0x4)
                0x00400000                _sfixed = .
 *(.vectors .vectors.* .vectors_default .vectors_default.*)
 .vectors.default
                0x00400000      0x168 startup_atsamv71j19b.o
                0x00400000                exception_table
 *(.isr_vector)
 *(.reset*)
 *(.after_vectors)

.text           0x00400168       0x20
                0x00400168                . = ALIGN (0x4)
 *(.glue_7t)
 .glue_7t       0x00400168        0x0 linker stubs
 *(.glue_7)
 .glue_7        0x00400168        0x0 linker stubs
 *(.gnu.linkonce.r.*)
 *(.ARM.extab* .gnu.linkonce.armextab.*)
                0x00400168                . = ALIGN (0x4)
 *(.init)
 .init          0x00400168        0x4 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
                0x00400168                _init
 .init          0x0040016c        0x8 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o
                0x00400174                . = ALIGN (0x4)
                0x00400174                __preinit_array_start = .
 *(.preinit_array)
                0x00400174                __preinit_array_end = .
                0x00400174                . = ALIGN (0x4)
                0x00400174                __init_array_start = .
 *(SORT_BY_NAME(.init_array.*))
 *(.init_array)
 .init_array    0x00400174        0x4 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
                0x00400178                __init_array_end = .
                0x00400178                . = ALIGN (0x4)
 *crtbegin.o(.ctors)
 *(EXCLUDE_FILE(*crtend.o) .ctors)
 *(SORT_BY_NAME(.ctors.*))
 *crtend.o(.ctors)
                0x00400178                . = ALIGN (0x4)
 *(.fini)
 .fini          0x00400178        0x4 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
                0x00400178                _fini
 .fini          0x0040017c        0x8 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o
                0x00400184                . = ALIGN (0x4)
                0x00400184                __fini_array_start = .
 *(.fini_array)
 .fini_array    0x00400184        0x4 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
 *(SORT_BY_NAME(.fini_array.*))
                0x00400188                __fini_array_end = .
 *crtbegin.o(.dtors)
 *(EXCLUDE_FILE(*crtend.o) .dtors)
 *(SORT_BY_NAME(.dtors.*))
 *crtend.o(.dtors)
                0x00400188                . = ALIGN (0x4)
                0x00400188                _efixed = .
                [!provide]                PROVIDE (__exidx_start = .)

.ARM.exidx
 *(.ARM.exidx* .gnu.linkonce.armexidx.*)
                [!provide]                PROVIDE (__exidx_end = .)
                0x00400188                . = ALIGN (0x4)
                0x00400188                _etext = .

.bss            0x20400000        0x0
```

```
                0x20400000                . = ALIGN (0x4)
                0x20400000                __bss_start__ = .
                0x20400000                _sbss = .
                0x20400000                _szero = .
 *(COMMON)
                0x20400000                . = ALIGN (0x4)
                0x20400000                __bss_end__ = .
                0x20400000                _ebss = .
                0x20400000                _ezero = .
                0x20400000                . = ALIGN (0x4)
                0x20400000                _end = .
                0x2043ffff                _ram_end_ = ((ORIGIN (ram) + LENGTH (ram)) - 0x1)
OUTPUT(dist/default/production/XC32_sam_library_example.X.production.elf elf32-littlearm)
LOAD stack
LOAD data_init

.ARM.attributes
                0x00000000       0x32
 .ARM.attributes
                0x00000000       0x1e /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crti.o
 .ARM.attributes
                0x0000001e       0x32 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
 .ARM.attributes
                0x00000050       0x32 build/default/production/_ext/1472/xc32_floats.o
 .ARM.attributes
                0x00000082       0x32 startup_atsamv71j19b.o
 .ARM.attributes

0x000000b4       0x32 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .ARM.attributes

0x000000e6       0x32 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .ARM.attributes

0x00000118       0x34 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .ARM.attributes

0x0000014c       0x32 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .ARM.attributes

0x0000017e       0x32 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)
 .ARM.attributes
                0x000001b0       0x1e /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtn.o

.comment        0x00000000       0x2f
 .comment       0x00000000       0x2f /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
                                 0x30  (size before relaxing)
 .comment       0x0000002f       0x30 build/default/production/_ext/1472/xc32_floats.o
 .comment       0x0000002f       0x30 startup_atsamv71j19b.o
 .comment
0x0000002f       0x30 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .comment
0x0000002f       0x30 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .comment
0x0000002f       0x30 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .comment
0x0000002f       0x30 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .comment
0x0000002f       0x30 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.debug_info     0x00000000     0x1666
 .debug_info    0x00000000      0x318 build/default/production/_ext/1472/xc32_floats.o
 .debug_info
0x00000318      0x2a2 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
```

```
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_info
0x000005ba      0x2a5 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_info
0x0000085f      0x623 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .debug_info
0x00000e82      0x535 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .debug_info
0x000013b7      0x2af /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.debug_abbrev   0x00000000      0x2d6
 .debug_abbrev  0x00000000       0x57 build/default/production/_ext/1472/xc32_floats.o
 .debug_abbrev
0x00000057       0x2b /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_abbrev
0x00000082       0x2e /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_abbrev
0x000000b0      0x117 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .debug_abbrev
0x000001c7       0xb8 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .debug_abbrev
0x0000027f       0x57 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.debug_aranges  0x00000000       0xb8
 .debug_aranges
                0x00000000       0x20 build/default/production/_ext/1472/xc32_floats.o
 .debug_aranges

0x00000020       0x20 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_aranges

0x00000040       0x20 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_aranges

0x00000060       0x28 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .debug_aranges

0x00000088       0x18 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .debug_aranges

0x000000a0       0x18 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.debug_ranges   0x00000000       0x70
 .debug_ranges  0x00000000       0x10 build/default/production/_ext/1472/xc32_floats.o
 .debug_ranges
0x00000010       0x10 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_ranges
0x00000020       0x10 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_ranges
0x00000030       0x40 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)

.debug_line     0x00000000      0x5ab
 .debug_line    0x00000000       0x73 build/default/production/_ext/1472/xc32_floats.o
 .debug_line
0x00000073       0xa7 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_line
0x0000011a      0x101 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_line
0x0000021b      0x1d1 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
```

```
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .debug_line
0x000003ec       0xba /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .debug_line
0x000004a6      0x105 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.debug_str      0x00000000        0x0
 .debug_str     0x00000000        0x0 build/default/production/_ext/1472/xc32_floats.o
 .debug_str
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_str
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_str
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .debug_str
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .debug_str
0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.debug_frame    0x00000000       0xa8
 .debug_frame   0x00000000       0x20 build/default/production/_ext/1472/xc32_floats.o
 .debug_frame
0x00000020       0x20 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
 .debug_frame
0x00000040       0x2c /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_frame
0x0000006c       0x3c /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)

.debug_loc      0x00000000       0x65
 .debug_loc
0x00000000       0x14 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
 .debug_loc
0x00000014       0x51 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)

.note.GNU-stack
                0x00000000        0x0
 .note.GNU-stack

0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
 .note.GNU-stack

0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
 .note.GNU-stack

0x00000000        0x0 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)

.stack          0x204000b8      0x100
 .stack         0x204000b8      0x100 stack

.bss%1          0x20400000       0x6c
 .bss
0x20400000       0x50 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(libc.o)
                0x20400000                program_invocation_name
                0x20400000                __progname_full
                0x20400004                __progname
                0x20400004                program_invocation_short_name
                0x20400008                __sysinfo
                0x2040000c                __hwcap
                0x20400010                __libc
 .bss           0x20400050       0x1c /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o
```

```
.bss%3          0x20400070       0x14
 .bss           0x20400070       0x14 build/default/production/_ext/1472/xc32_floats.o
                0x20400070                result
                0x20400078                myDouble
                0x20400080                myFloat

.bss%4          0x2040006c        0x4
 .bss
0x2040006c        0x4 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)

.bss%5          0x20400084        0x4
 .bss
0x20400084        0x4 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__environ.o)
                0x20400084                _environ
                0x20400084                __environ
                0x20400084                environ
                0x20400084                ___environ

.eh_frame%z1    0x00000000        0x0
 .eh_frame      0x00000000        0x0 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o

.text%6         0x00400188      0x110
 .text          0x00400188      0x110 startup_atsamv71j19b.o

.text.Reset_Handler%7
                0x00400298       0x8c
 .text.Reset_Handler
                0x00400298       0x8c startup_atsamv71j19b.o
                0x00400298                Reset_Handler

.text.__pic32c_data_initialization_impl%8
                0x00400324       0x68
 .text.__pic32c_data_initialization_impl

0x00400324       0x68 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_0_1.o)
                0x00400324                __pic32c_data_initialization_impl
                0x00400324                __copy_needed

.text.__libc_start_init%9
                0x0040038c       0x5c
 .text.__libc_start_init

0x0040038c       0x5c /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
                0x0040038c                __libc_init_array

.dinit%10       0x004003e8       0x50
 .dinit         0x004003e8       0x50 data_init

.text%11        0x00400438       0x40
 .text          0x00400438       0x40 /Applications/microchip/xc32/
v4.30/bin/bin/../../lib/gcc/pic32c/8.3.1/thumb/v7e-m+dp/hard/crtbegin.o

.text.main%12   0x00400478       0x30
 .text.main     0x00400478       0x30 build/default/production/_ext/1472/xc32_floats.o
                0x00400478                main

.text.__pic32c_data_initialization%13
                0x004004a8        0x4
 .text.__pic32c_data_initialization

0x004004a8        0x4 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libpic32c.a(data_init_wrapper.o)
                0x004004a8                __pic32c_data_initialization

.text.Dummy_Handler%14
                0x004004ac        0x2
 .text.Dummy_Handler
                0x004004ac        0x2 startup_atsamv71j19b.o
                0x004004ac                TWIHS0_Handler
                0x004004ac                SVCall_Handler
                0x004004ac                Reserved18_Handler
                0x004004ac                TC0_CH2_Handler
                0x004004ac                Reserved69_Handler
                0x004004ac                TC3_CH1_Handler
                0x004004ac                Reserved38_Handler
                0x004004ac                UART2_Handler
                0x004004ac                TRNG_Handler
                0x004004ac                HardFault_Handler
                0x004004ac                TC2_Handler
                0x004004ac                Reserved12_Handler
                0x004004ac                SysTick_Handler
                0x004004ac                AFEC1_Handler
                0x004004ac                PendSV_Handler
                0x004004ac                TC0_CH1_Handler
                0x004004ac                TC7_Handler
                0x004004ac                Reserved9_Handler
                0x004004ac                QSPI_Handler
                0x004004ac                NonMaskableInt_Handler
                0x004004ac                ISI_Handler
                0x004004ac                MCAN0_INT0_Handler
                0x004004ac                Reserved41_Handler
                0x004004ac                TC9_Handler
                0x004004ac                RSWDT_Handler
                0x004004ac                USART0_Handler
                0x004004ac                TC1_CH2_Handler
                0x004004ac                RTT_Handler
                0x004004ac                USBHS_Handler
                0x004004ac                IXC_Handler
                0x004004ac                CCF_Handler
                0x004004ac                Reserved55_Handler
                0x004004ac                PIOB_Handler
                0x004004ac                RTC_Handler
                0x004004ac                UsageFault_Handler
                0x004004ac                TC6_Handler
                0x004004ac                Reserved37_Handler
                0x004004ac                WDT_Handler
                0x004004ac                TC4_Handler
                0x004004ac                TC1_Handler
                0x004004ac                UART1_Handler
                0x004004ac                TC3_Handler
                0x004004ac                Dummy_Handler
                0x004004ac                TWIHS1_Handler
                0x004004ac                TC3_CH2_Handler
                0x004004ac                XDMAC_Handler
                0x004004ac                TC2_CH0_Handler
                0x004004ac                PWM1_Handler
                0x004004ac                USART1_Handler
                0x004004ac                PWM0_Handler
                0x004004ac                PIOA_Handler
                0x004004ac                AFEC0_Handler
                0x004004ac                CCW_Handler
                0x004004ac                TC3_CH0_Handler
                0x004004ac                SSC_Handler
                0x004004ac                TC11_Handler
                0x004004ac                TC1_CH0_Handler
                0x004004ac                Reserved21_Handler
                0x004004ac                Reserved17_Handler
                0x004004ac                ACC_Handler
                0x004004ac                PMC_Handler
                0x004004ac                GMAC_Handler
                0x004004ac                TC2_CH2_Handler
                0x004004ac                Reserved42_Handler
                0x004004ac                MCAN0_INT1_Handler
                0x004004ac                GMAC_Q5_Handler
                0x004004ac                SUPC_Handler
                0x004004ac                TC2_CH1_Handler
                0x004004ac                TC8_Handler
                0x004004ac                EFC_Handler
                0x004004ac                GMAC_Q1_Handler
                0x004004ac                GMAC_Q2_Handler
                0x004004ac                TC1_CH1_Handler
                0x004004ac                TC10_Handler
                0x004004ac                GMAC_Q3_Handler
                0x004004ac                PIOD_Handler
                0x004004ac                MLB_Handler
                0x004004ac                Reserved46_Handler
                0x004004ac                Reserved70_Handler
                0x004004ac                Reserved62_Handler
                0x004004ac                UART0_Handler
                0x004004ac                TC0_CH0_Handler
                0x004004ac                Reserved45_Handler
                0x004004ac                Reserved15_Handler
                0x004004ac                GMAC_Q4_Handler
                0x004004ac                FPU_Handler
                0x004004ac                BusFault_Handler
                0x004004ac                RSTC_Handler
                0x004004ac                TC0_Handler
                0x004004ac                MemoryManagement_Handler
                0x004004ac                Reserved54_Handler
                0x004004ac                DebugMonitor_Handler
                0x004004ac                TC5_Handler
                0x004004ac                ICM_Handler
                0x004004ac                AES_Handler
                0x004004ac                DACC_Handler

.text.__dummy%15
                0x004004ae        0x2
 .text.__dummy
0x004004ae        0x2 /Applications/microchip/xc32/v4.30/bin/bin/../../lib/gcc/pic32c/
8.3.1/../../../../pic32c/lib/thumb/v7e-m+dp/hard/libc-musl.a(__libc_start_main.o)
                0x004004ae                __init_host
                0x004004ae                __init_stdio
```

8. Linker Command-Line Interface

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

8.1 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. The general form of a command to run the linker looks like the following.

```
xc32-ld [options] files
```

The linker *options* are case sensitive and are described in sections following. The *files* are one or more object files (.o extension) or library archive files (.a extension) which will be linked together to form the final output.

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 library archive libpic32.a.

When linking a C application, there are typically several library archives that 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.

8.2 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
```

The driver *`options`* are case sensitive. The *`files`* are one or more object files (`.o` extension) or library archive files (`.a` extension) which will be linked together to form the final output.

To pass a linker option to the linker without having to invoke the linker directly, use the driver's `-Wl,`*`option`* option. For example, the following will pass the `--defsym` option directly to the linker.

```
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 options relating to ISA mode and floating-point support etc., to select the appropriate multilib permutation. See the C compiler User's Guide relevant for your target device for more information on the C compiler's multilib feature.

8.3 Options That Control Output File Creation

8.3.1 Start-group/End-group Options

The `--start-group` *`archives`*`, --end-group` option marks the start and end of a group of library archives.

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

8.3.2 D: Assign Space to Common Symbols Option

The `-d` option (or either of its aliases `-dc` or `-dp`) force common symbols to be defined. Space is assigned to common symbols even if a relocatable output file is specified (with `-r`). The script command `FORCE_COMMON_ALLOCATION` has the same effect.

8.3.3 Defsym Option

The `--defsym` *`sym=expr`* option defines a symbol.

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

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

8.3.4 X: Discard Local Symbols Option

The `-x` option or its alias `--discard-all` discards all local symbols from the output.

8.3.5 X: Discard Temporary Local Symbols Option

The `-X` option or its alias `--discard-locals` discards temporary local symbols from the output.

8.3.6 Fill Option

The `--fill=option` option fills unused program memory. It is currently available for only PIC32M devices. The general form of this option is:

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

where *`address`* and *`end_address`* 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. The *`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.

8.3.7 Gc-sections Option

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

This option can sometimes interfere with the debugging experience associated with a project, because sections containing functions can be removed from the linker while the debugging information for these deleted functions remains in the ELF file.

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))`.

8.3.8 Library Option

The `--library` *`name`* `(-l` *`name`*`)` option searches for a library archive with the name *`name`*.

This option adds library archive file called *`name`* to the list of files to link. This option may be used any number of times. The linker will search its path-list for occurrences of `lib`*`name`*`.a` for every *`name`* specified. The linker will search a library 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 library archive file is not recognized, the linker will ignore it. Therefore, a version mismatch between libraries and the linker may result in “undefined symbol” errors.

8.3.9 L: Library Path Option

The -L *dir* option or its alias --library-path *dir* adds *dir* to search path the linker will use to find libraries.

This option adds 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 libraries specified using -l options, regardless of the order in which the options appear. The library paths can also be specified in a link script with the SEARCH_DIR command. Directories specified this way are searched at the point in which the linker script appears in the command line.

8.3.10 Nodefaultlibs Option

The -nodefaultlibs option will prevent the standard system libraries being linked into the project. Only the libraries you specify are passed to the linker.

8.3.11 Nostartfiles Option

The -nostartfiles option will prevent the runtime startup modules from being linked into the project.

8.3.12 Nostdlib Option

The -nostdlib option will prevent the standard system startup files and libraries being linked into the project. No startup files and only the libraries you specify are passed to the linker.

8.3.13 O: Specify Output File

The -o option specifies the base name and directory of the output file.

The option -o main.elf, for example, will place the generated output in a file called main.elf. The name of an existing directory can be specified with the file name, for example -o build/main.elf, so that the output file will appear in that directory.

You cannot use this option to change the type (format) of the output file.

8.3.14 P: Specify Processor Option

The --p=*processor* option specifies the target processor for the link step, for example, --p=32MX795F512L.

8.3.15 R: Generate Relocatable Output Option

The -r option or its aliases -i or --relocatable generates a relocatable output.

This option generates an output file that has not had symbols resolved and 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.

8.3.16 Retain-symbols-file Option

The --retain-symbols-file *file* option keeps only those symbols listed in *file*.

This option retains only the symbols listed in the file *file*, discarding all others. The *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 the `-s` and `-S` options.

8.3.17 Section-start Option

The `--section-start sectionname=`*`org`* option locates 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. The *org* argument 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`*.

8.3.18 T: Specify Linker Script Option

The `-T` *`file`* option or its alias `--script` *`file`* reads commands from the specified linker script.

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

8.3.19 S: Strip All Symbols Option

The `-s` option or its alias `--strip-all` strip all symbols from the output file.

8.3.20 S: Strip Debug Symbols Option

The `-S` option or its alias `--strip-debug` strip debug symbols (but not all symbols) from the output file.

8.3.21 Tbss Option

The `-Tbss` *`address`* option sets the address of the `.bss` section.

Use *`address`* as the starting address for the bss segment of the output file. The *`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.

8.3.22 Tdata Option

The `-Tdata` *`address`* option sets the address of the `.data` section.

Use *`address`* as the starting address for the data segment of the output file. The *`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.

8.3.23 Ttext Option

The `-Ttext` *`address`* option sets the address of the `.text` section.

Use *`address`* as the starting address for the text segment of the output file. The *`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.

8.3.24 U: Insert Undefined Symbol Option

The `-u` *`symbol`* option or its alias `--undefined` *`symbol`*inserts an undefined reference to *`symbol`* into the output file. Doing this may, for example, trigger linking of additional modules from standard libraries. The `-u` option may be repeated with different option arguments to enter additional undefined symbols.

8.3.25 No-undefined Option

The `--no-undefined` option reports unresolved symbol references in regular object files.

8.3.26 Wrap Option

The `--wrap` *`symbol`* option uses wrapper functions for *`symbol`*.

Any undefined references to symbol will be resolved to `__wrap_`*`symbol`*. (Note the two leading underscores used with these symbols.) Any undefined references to `__real_`*`symbol`* will be resolved to *`symbol`*. This can be used to provide a wrapper for a system function. The wrapper function should be called `__wrap_`*`symbol`*. If it wishes to call the system function, it should call `__real_`*`symbol`*.

Here is a trivial example:

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

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

8.4 Options That Control Run-Time Initialization

8.4.1 Data-init Option

The `--data-init` option creates 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. This is the default action if no option is specified.

The `--no-data-init` form of this option selects an implementation of the C start-up module in `libpic32.a` that contains weak definitions for the data initialization routines. You should override these in your code as required to prevent an error at runtime. For PIC32M targets, override the routine `__pic32_data_init`, for PIC32C/SAM devices, override the routine `__pic32c_data_initialization`. Alternatively, remove the call to the data-init function from the device startup code.

8.4.2 `--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
```

8.4.3 `--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
```

8.5 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 prebuilt 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.

8.5.1 Mips32r2 Option

The `-mips32r2` option requests that code be built using the MIPS32 Release 2 architecture.

8.5.2 Micromips Option

The `-mmicromips` option informs the compiler to generate code using the microMIPS™ instructions. This feature is available only in the PRO edition.

When your device is configured to boot into the microMIPS compressed Instruction Set Architecture (ISA) mode (for example, if `#pragma config BOOTISA=MICROMIPS` was specified), the `-mmicromips` option should be used when linking to select microMIPS startup code.

The `-mno-micromips` form of this option informs the compiler to use MIPS32 instructions.

The ISA can also be indicated on a per-function basis through the `micromips` and `nomicromips` attributes.

On PIC32M devices, bit 0 of the program counter indicates the ISA mode. When this bit is clear, the device is running in MIPS32 mode. When this bit is set, the device is running in either MIPS16 or microMIPS mode, depending on the core of the selected device. This means that if you execute a hard-coded jump, bit 0 must be set to the appropriate value for the destination function. Hard-coded jumps are most commonly seen when jumping from a bootloader to a bootloaded application.

8.5.3 Mips16 Option

The `-mips16` option generates MIPS16 code, suitable for MIPS16 or microMIPS Instruction Set Architecture (ISA) mode. This option is only available in the PRO edition.

The `-mno-mips16` option (note the slightly different 'no' form of this option, which includes an additional 'm' character) will have all code built using the MIPS32 ISA.

On PIC32M devices, bit 0 of the program counter indicates the ISA mode. When this bit is clear, the device is running in MIPS32 mode. When this bit is set, the device is running in either MIPS16 or microMIPS mode, depending on the core of the selected device. This means that if you execute a hard-coded jump, bit 0 must be set to the appropriate value for the destination function. Hard-coded jumps are most commonly seen when jumping from a bootloader to a bootloaded application.

8.5.4 Soft-float Option

The **`-msoft-float`** option instructs the linker to link in libraries that contain full software floating-point support.

8.5.5 No-float Option

The `-mno-float` option instructs the linker to not link in libraries that contain software floating-point code.

8.6 Options that Control Informational Output

8.6.1 Check-sections Option

The `--check-sections` option requests that the linker check section addresses for overlap, producing a suitable error message in such an event. This is the default action if no option is specified. If this option is used, it will force the linker to check for section overlap relocatable links, which is not usually performed.

The `--no-check-sections` form of this option requests that the linker does not perform any check for overlap of section addresses. This might be used for diagnosing memory allocation issues.

8.6.2 Help Option

The `--help` option prints a summary of the command-line options on the standard output and then exits.

8.6.3 No-warn-mismatch Option

The `--no-warn-mismatch` option tells the linker that it should not issue any error messages when encountering input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for a different endiannesses.

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.

8.6.4 Report-mem Option

The `--report-mem` option prints a summary of memory usage to standard output during the link. This report also appears in the link map.

8.6.5 T: Print Input Files Option

The `-t` option or its alias `--trace` print the names of the input files as `xc32-ld` processes them.

8.6.6 Y: Print File Using Symbol Option

The `-y` *`symbol`* option or its alias `--trace-symbol` *`symbol`* prints the name of each linked file in which *`symbol`* appears. This option may be used any number of times and is useful when you have an undefined symbol in your build but do not know where the reference is coming from.

8.6.7 V: Version Options

The `-v` option prints version information before it builds the input files. The `--version` option prints the version as well as other license information. The `-V` option (uppercase) prints the version information and lists supported emulations.

8.6.8 Warn-common Option

The `--warn-common` option warns 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
```

8.6.9 Warn-once Option

The `--warn-once` option warns only once for each undefined symbol, rather than once per module that refers to it.

8.6.10 Warn-section-align Option

The `--warn-section-align` option warns if the start of a 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.

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

8.7 Options that Modify the Link Map Output

8.7.1 Cref Option

The `--cref` option outputs a cross reference table.

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

8.7.2 M: Print Map File Option

The `-M` option or its alias `--print-map` prints a map file to standard output. A link map provides information about the link, including the following:

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

8.7.3 Map Option

The `-Map` *`file`* option writes a link map to the specified file.

See the description of the `-M` option (8.7.2. M: Print Map File Option) for information on what is contained in a link map.

9. Linker Scripts

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.

9.1 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.

Typically, two linker scripts are used to fully define how a program should be linked. The main linker script will have an extension that is or begins with x, for example elf32pic32c.x or elf32pic32mx.xe and copies of these are located in the *family*/lib/ldscripts directories of your compiler distribution. Device-specific fragments of the linker scripts have a .ld extension and copies of these are located in the *family*/lib/proc/*device* directories.

9.2 Command Line Information

Linker scripts are specified on the command line using either the -T option or the --script option (see 3.6. Options That Control Output File Creation). For example:

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

If the linker is invoked through xc32-gcc, use the driver's -Wl, option to pass the linker option directly to the linker. For example:

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

9.3 Default Linker Script

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 32-bit devices. It uses an INCLUDE directive to include the device-specific memory regions.

The default linker script is appropriate for most 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 shipped. Note that this file is only a copy of the default linker script. The script that the linker uses is internal to the linker. If the -mdfp and -mprocessor option is used with an XC32 application to indicate an alternate device family pack (DFP), the linker script from that DFP will be used instead.

For PIC32MX Devices

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 portions of the linker script are located in /pic32mx/lib/proc/*device*/, where *device* is the device name specified with the -mprocessor compilation-driver (xc32-gcc) option.

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` option.

Note: The Compiler guide relevant for you target device 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 9-1. PIC32 Reserved, Standard Section Names in Default Linker Script

Section Name	Generated by	Final Location	Default linker-script MEMORY region	Implied attributes
`.reset`	Reset handler	Executable boot-code segment	`kseg0_boot_mem`	code
`.bev_excpt`	BEV-Exception handler	Executable boot-code segment	`kseg0_boot_mem`	code
`.app_excpt`	General-Exception handler	Executable boot-code segment	`kseg0_boot_mem`	code
`.vector_n`	Interrupt Vector n	Executable boot-code segment	`kseg0_boot_mem`	code
`.startup`	C startup code	Executable boot-code segment	`kseg0_boot_mem`	code
`.text`	Compiler- or assembler-generated instructions	Executable code segment	`kseg0_program_mem`	code
`.rodata`	Strings and C data declared const	Read-only data segment	`kseg0_program_mem`	code
`.sdata2`	Small initialized constant global and static data	Read-only data segment	`kseg0_program_mem`	data
`.sbss2`	Uninitialized constant global and static data (i.e., variables which will always be zero)	Read-only data segment	`kseg0_program_mem`	data

`.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`
				data

`.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`
				data

`.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`
				data

`.sbss`	Uninitialized variables <=n bytes (compiled `-Gn`). Used for gp-relative addressing.	Small zero-filled segment	`kseg1_data_mem`	data
`.bss`	Uninitialized larger variables	Zero-filled segment	`kseg1_data_mem`	data
`.heap`	Heap used for dynamic memory	Reserved by linker script	`kseg1_data_mem`	data

...........continued

Section Name	Generated by	Final Location	Default linker-script MEMORY region	Implied attributes
.stack	Minimum space reserved for stack	Reserved by linker script	kseg1_data_mem	data
.ramfunc	RAM-functions, copied from program memory to data memory at C startup	Initialized data segment	kseg1_data_mem & kseg0_program_mem	data
.reginfo .stab* .debug*	Debug information	Not in load image	n/a	info
.line	DWARF debug information	Not in load image	n/a	info
.comment	#ident / .ident directive	Not in load image	n/a	info

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.

For PIC32C/SAM Devices

A copy of the default linker script is located at /pic32c/lib/ldscripts/elf32pic32c.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 portions of the linker script are located in /pic32c/lib/proc/*device*/, where *device* is the device name specified with the -mprocessor compilation-driver (xc32-gcc) option.

9.4 Using Custom Linker Scripts in MPLAB X IDE Projects

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 device-specific fragment of the linker script file (see 9.3. Default Linker Script) into your application's project directory. This file will have a .ld extension and should be added to your project. It should appear in the project tree under “Linker Files.” Upon linking your project, the linker prints the name and path of the linker script that is currently being used. Use this to confirm that your script is being selected.

Customizations that you make to your new *.ld file will be reflected in subsequent builds.

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.

9.5 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.

9.5.1 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 contains, 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 device family reference manual relevant for your target device for a description of the Virtual-to-Physical Fixed Memory Mapping. In addition, the family reference manual describes the device's 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.

9.5.2 Commands Dealing with Files

Several linker script commands deal with files.

Include Command

The `INCLUDE` *`filename`* command 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 Command

The `INPUT(`*`file, file, ...`*`)` or `INPUT(`*`file file ...`*`)` 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 the `-L` option in 8.3.9. L: Library Path Option.

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 Command

The `GROUP(`*`file, file, ...`*`)` or `GROUP(`*`file file ...`*`)` 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 8.3.1. Start-group/End-group Options.

As_needed Command

The `AS_NEEDED(file, file, ...)` or `AS_NEEDED(file file ...)` command can appear only inside of the `INPUT` or `GROUP` commands, among other filenames. The files listed will be handled as if they appear directly in the `INPUT` or `GROUP` commands; however, if the file is an ELF shared library, that will be added only when it is actually needed. This construct essentially enables the `--as-needed` command-line option for all the files listed with this command and then restores previous setting afterwards.

Optional Command

The `OPTIONAL(file, file, ...)` 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 Command

The `OUTPUT(filename)` command names the output file. Using this command in the linker script has a similar effect to using the `-o filename` option on the command line (see 8.3.13. O: Specify Output File). If both are used, the command line option takes precedence.

Search_dir Command

The `SEARCH_DIR(path)` command adds *`path`* to the list of paths where the linker looks for archive libraries. Using this command in the linker script has a similar effect to using the `-L path` option on the command line (see 8.3.9. L: Library Path Option). If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

Startup Command

The `STARTUP(filename)` command is just like the `INPUT` command, except that *`filename`* will become the first input file to be linked, as though it were specified first on the command line.

9.5.3 Assigning Values to Symbols

A value may be assigned to a symbol in a linker script. This will define the symbol as a global symbol.

9.5.3.1 Simple Assignments

A symbol may be assigned using any of the C-style assignment operators:

```
symbol = expression ;
symbol += expression ;
symbol -= expression ;
symbol *= expression ;
symbol /= expression ;
symbol <<= expression ;
symbol >>= expression ;
symbol &= expression ;
symbol |= expression ;
```

The first of the above examples will define *`symbol`* to have the value of *`expression`*. In the other examples, *`symbol`* must already be defined, and the value will be adjusted according to the results of the operation.

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 9.6. 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 9.6.6. 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.

9.5.3.2 Provide Command

The `PROVIDE` command defines a referenced symbol that is not otherwise defined.

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, the C standard indicates that `etext` is a valid function name. 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 = .);
   }
}
```

If the symbol `etext` is defined by the program, that definition is used by the linker. If it is not defined by the program, then the definition provided by the linker script is used. Compare this with the definition for `_etext` (with a leading underscore) in the linker script, which will trigger an error if it is also defined in the program.

The default linker script used with PIC32M devices makes use of the `PROVIDE` command to define the default `_min_stack_size`, `_min_heap_size`, and `_vector_spacing` symbol values, for example:

```
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
PROVIDE(_vector_spacing = 0x00000001);
```

9.5.4 Memory Command

The `MEMORY` command overrides the linker’s default allocation of all available memory.

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 KB, and the other starting at 0x800 for 2 KB. The linker will place into the rom memory region every section which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the ram memory region.

```
  MEMORY
{
   rom (rx)  : ORIGIN = 0, LENGTH = 48K
   ram (!rx) : org = 0x800, l = 2K
}
```

Once a memory region is defined, the linker can be directed to place specific output sections into that memory region by using the >region output section attribute. For example, to specify a memory region named mem, use >mem in the output section definition. If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message.

9.5.5 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 9.5.6. Other Linker Script Commands)
- a symbol assignment (see 9.5.3. Assigning Values to Symbols)
- an output section description
- an overlay description

The ENTRY command and symbol assignments are permitted inside the SECTIONS command where they may use the location counter (see 9.6.3. The Location Counter). This can also make the linker script easier to understand because those commands can be used at meaningful points in the layout of the output file.

Output section descriptions and overlay descriptions are described below.

If a SECTIONS command does not appear in the linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

9.5.5.1 Input Section Description

Input section descriptions tell the linker how to map the input files into the memory layout and represent the majority of the actions performed by a SECTION command.

The input section description is the most basic linker script operation and 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, the EXCLUDE_FILE command 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.

9.5.5.2 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 wildcards in the order in which they are seen during the link. This can be changed by using the SORT keyword, which appears before a wildcard pattern in parentheses (e.g., SORT(.text*)). When the SORT keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

To verify where the input sections are going, use the -M linker option to generate a map file. The map file shows precisely how input sections are mapped to output sections.

This example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all .text sections in .text and all .bss sections in .bss. The linker will place the .data section from all files beginning with an upper case character in .DATA; for all other files, the linker will place the .data section in .data.

```
  SECTIONS
{
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

9.5.5.3 Input Section for Common Symbols

A special notation is needed for common symbols, because common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named `COMMON`.

File names may be used with the `COMMON` section just as with any other input sections. This will place common symbols from a particular input file in one section, while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the `.bss` section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

If not otherwise specified, common symbols will be assigned to section `.bss`.

9.5.5.4 Input Section Example

The following example is a complete linker script. It tells the linker to read all of the sections from file `all.o` and place them at the start of output section `outputa` which starts at location `0x10000`. All of section `.input1` from file `foo.o` follows immediately, in the same output section. All of section `.input2` from `foo.o` goes into output section `outputb`, followed by section `.input1` from `foo1.o`. All of the remaining `.input1` and `.input2` sections from any files are written to output section `outputc`.

```
SECTIONS
{
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

9.5.5.5 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 need to make use 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 9.5.3. Assigning Values to Symbols)

- an input section description (see 9.5.5.1. Input Section Description)
- data values to include directly (see 9.5.5.7. Output Section Data)

9.5.5.6 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 9.6. 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) }
```

The above command works because `ALIGN` returns the current location counter aligned upward to the specified value.

Specifying *address* for a section will change the value of the location counter.

9.5.5.7 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.

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 9.5.5.9. Output Section Attributes), but it only affects the part of the section following the `FILL` command, rather than the entire section. If both are used, the `FILL` command takes precedence.

9.5.5.8 Output Section Discarding

The linker will not create an output section which does not have any contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. For example:

```
.foo { *(.foo) }
```

will only create a `.foo` section in the output file if there is a `.foo` section in at least one input file.

If anything other than an input section description is used as an output section command, such as a symbol assignment, then the output section will always be created, even if there are no matching input sections.

The special output section name `/DISCARD/` may be used to discard input sections. Any input sections which are assigned to an output section named `/DISCARD/` are not included in the output file.

9.5.5.9 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]
```

The *`name`*, *address* and *`output-section-command`* components have already been described. In the following sections, the remaining section attributes will be described.

9.5.5.9.1 Output Section Type

Each output section may have a type. The type is a keyword in parentheses.

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) : { ... }
  ...
}
```

The following types are defined.

Noload Type

The `NOLOAD` type marks a section as not loadable, so that it will not be loaded into memory when the program is run.

Dsetc, Copy, Info, and Overlay Types

The DSECT, COPY, INFO, and OVERLAY 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.

9.5.5.9.2 Output Section LMA

Every section has a virtual address (VMA) and a load address (LMA). The address expression that 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 9.5.4. 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 C compiler user's guide relevant for your target device for more information on the startup code provided with the compiler.

9.5.5.9.3 Output Section Region

A section can be assigned to a previously defined region of memory by using >*region*. See 9.5.4. Memory Command.

Here is a simple example:

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

9.5.5.9.4 Output Section Fill

A fill pattern can be set for an entire section by using =*fillexp*. The *fillexp* is 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 9.5.5.7. Output Section Data.

Here is a simple example:

```
SECTIONS { .text : { *    (.text) } =0x9090 }
```

9.5.5.10 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));
```

9.5.6 Other Linker Script Commands

There are several other linker script commands, which are described below.

Assert Command

The `ASSERT(`*`exp, message`*`)` command ensures that *`exp`* is non-zero. If it is zero, then exit the linker with an error code, and print *`message`*.

Entry Command

The `ENTRY(`*`symbol`*`)` command specifies *`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 Command

The `EXTERN(`*`symbol symbol ...`*`)` command forces *`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 Command

The `FORCE_COMMON_ALLOCATION` 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 Command

The `NOCROSSREFS(`*`section section ...`*`)` 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 Command

The `OUTPUT_ARCH(`*`bfdarch`*`)` command specifies a particular output machine architecture. The *`bfdarch`* value is `pic32mx` for Microchip PIC32 MCUs, for example. Other values for ARm-based devices include `armv5tej`, `armv6s-m`, `armv7`, `armv7e-m`, `armv8-m.base` and `armv8-m.main`.

Output_format Command

The `OUTPUT_FORMAT(`*`format_name`*`)` 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 Command

The `TARGET(`*`format_name`*`)` 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.

9.6 Expressions in Linker Scripts

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as 32-bit integers.

You can use and set symbol values in expressions.

The linker defines several special purpose built-in functions for use in expressions.

9.6.1 Constants

All constants are integers.

As in C, the linker considers an integer beginning with `0` to be octal, and an integer beginning with `0x` or `0X` to be hexadecimal. The linker considers other integers to be decimal.

In addition, you can use the suffixes `K` and `M` to scale a constant by 1024 or 1024*1024 respectively. For example, the following all refer to the same quantity:

```
_fourk_1 = 4K;
_fourk_2 = 4096;
_fourk_3 = 0x1000;
```

9.6.2 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword by surrounding the symbol name in double quotes:

```
"SECTION" = 9;
"with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, `A-B` is one symbol, whereas `A - B` is an expression involving subtraction.

9.6.3 The Location Counter

The special linker variable represented by a dot (`.`) always contains the current output location counter. Since the `.` variable always refers to a location in an output section, it may only appear in an expression within a `SECTIONS` command. In expressions, the `.` symbol may appear anywhere an ordinary symbol is permitted.

Assigning a value to `.` will cause the value of the location counter to be changed and affect the location of any subsequent output. 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.

The `.` variable 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 the `.` variable can be used as an absolute address. If the `.` variable 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 the `.` variable backwards). The `.data` section will start at `0x500` and it will have an extra `0x600` bytes worth of space after the end of the values from the `.data` input sections and before the end of the `.data` output section itself.

9.6.4 Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels:

Table 9-2. 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

9.6.5 Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and lengths of memory regions, in order to do any linking at all. These values are computed as soon as possible when the linker reads in the linker script.

However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

The sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation.

Some expressions, such as those depending upon the location counter (`.`), must be evaluated during section allocation.

If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following:

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

will cause the error message `non-constant expression for initial address`.

9.6.6 The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.

The position of the expression within the linker script determines whether it is absolute or relative. An expression which appears within an output section definition is relative to the base of the output section. An expression which appears elsewhere will be absolute.

A symbol set to a relative expression will be relocatable if you request relocatable output using the `-r` option. That means that a further link operation may change the value of the symbol. The symbol's section will be the section of the relative expression.

A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.

You can use the built-in function `ABSOLUTE` to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section `.data`:

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

If `ABSOLUTE` were not used, `_edata` would be relative to the `.data` section.

9.6.7 Built-in Functions

The linker script language includes a number of built-in functions for use in linker script expressions.

9.6.7.1 Absolute Built-in Function

The `ABSOLUTE(`*`exp`*`)` built-in function returns 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.

9.6.7.2 Addr Built-in Function

The `ADDR(`*`section`*`)` built-in function returns the absolute address (a VMA) of the named section. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS { ...
  .output1 :
   {
```

```
   start_of_output_1 = ABSOLUTE(.);
   ...
   }
  .output :
   {
   symbol_1 = ADDR(.output1);
   symbol_2 = start_of_output_1;
   }
...
}
```

9.6.7.3 Align Built-in Function

The `ALIGN(`*`exp`*`)` built-in function returns the location counter (`.`) aligned to the next exp boundary. The argument, *`exp`*, must be an expression whose value is a power of two. This is equivalent to

```
(. + exp - 1) & ~(exp - 1)
```

`ALIGN` built-in function doesn't change the value of the location counter; it just performs 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 9.5.5. Sections Command). The second use of `ALIGN` is used to define the value of a symbol.

The built-in function `NEXT` is closely related to `ALIGN`.

9.6.7.4 Block Built-in Function

The `BLOCK(`*`exp`*`)` built-in function is a synonym for `ALIGN`, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

9.6.7.5 Defined Built-in Function

The `DEFINED(`*`symbol`*`)` built-in function return 1 if *`symbol`* is in the linker global symbol table and is defined; otherwise it returns 0. You can use this function to provide default values for symbols. For example, the following script fragment shows how to set a global symbol `begin` to the first location in the `.text` section, but if a symbol called `begin` already existed, its value is preserved:

```
SECTIONS { ...
  .text : {
   begin = DEFINED(begin) ? begin : . ;
   ...
  }
  ...
}
```

9.6.7.6 Keep Built-in Function

The `KEEP(`*`section`*`)` built-in function can be used to prevent a section from being eliminated as part of link-time garbage collection (`--gc-sections`). This is accomplished by surrounding an input section's wildcard entry with `KEEP()`, as in `KEEP(*(.init))` or `KEEP(SORT_BY_NAME(*)(.ctors))`.

9.6.7.7 Loadaddr Built-in Function

The `LOADADDR(`*`section`*`)` built-in function returns the absolute LMA of the named section. This is normally returns the same address as `ADDR`, but it may be different if the `AT` attribute is used in the output section definition (see 9.5.5. Sections Command).

9.6.7.8 Max Built-in Function

The `MAX(exp1, exp2)` built-in function returns the maximum value of the two expressions *exp1* and *exp2*.

9.6.7.9 Min Built-in Function

The `MIN(exp1, exp2)` built-in function returns the minimum value of the two expressions *exp1* and *exp2*.

9.6.7.10 Next Built-in Function

The `NEXT(exp)` built-in function return the next unallocated address that is a multiple of *exp*. This function is equivalent to `ALIGN()` *exp*.

9.6.7.11 Sizeof Built-in Function

The `SIZEOF(section)` built-in function 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);
...
}
```

10. Linker Processing

How the MPLAB XC32 Object Linker (`xc32-ld`) builds an application from input files is discussed here.

10.1 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

10.1.1 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 9. Linker Scripts for more details.

Next the linker opens all of the input object files. Each input file is checked to make sure the object format is compatible. If the object format is not compatible, an error is generated. The contents of each input file are then loaded into internal data structures. Typically each input file will contain multiple sections of code or data. Each section contains a list of relocation entries which associate locations in a section's raw data with relocatable symbols.

10.1.2 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 10.2. Linker Allocation.

10.1.3 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 10.3. Global and Weak Symbols. If any external symbol reference remains undefined, an error is generated.

10.1.4 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. The `.dinit` section contains an initialization template that is interpreted by the C run-time library. For more information about initialized data, see 10.4. Initialized Data.

10.1.5 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.

10.1.6 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 `–Map` linker 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. An additional memory-usage report by module is included, which shows the memory usage of each input object in the final ELF output. For more information about the link map, see 7.3.5. Map File.

10.2 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.

10.2.1 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 wildcard 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.

10.2.2 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:

- The section map specifies an absolute address for an output section.
- 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.

10.2.3 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.

Section attributes affect memory allocation as described below. For a general discussion of section attributes, see 16.1. Assembler Directives that Define Sections.

10.2.3.1 Allocation of Code-attributed Sections

The `code` section 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

10.2.3.2 Allocation of Data-attributed Sections

The `data` section attribute specifies that a section should be allocated as initialized storage in data memory, as defined by regions `kseg0_data_mem` and `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 64 KB of data memory
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1

10.2.3.3 Allocation of Bss-attributed Sections

The `bss` section attribute specifies that a section should be allocated as uninitialized storage in data memory, as defined by region `kseg0_data_mem` and `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 64 KB of data memory
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1

10.2.3.4 Allocation of Persist-attributed Sections

The `persist` section attribute specifies that a section should be allocated as persistent storage in data memory, as defined by region `kseg0_data_mem` and `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 64 KB of data memory
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1

10.3 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.

The names of weak symbols share the same name space as those for global symbols, but are handled differently. Multiple definitions of a weak symbol are permitted. If a weak definition is not found among the input files, archives are not searched and a value of 0 is assumed for all references to the weak symbol. A global symbol definition of the same name will take precedence over a weak definition (or the lack of one). In essence, weak symbols are considered optional and may be replaced by global symbols, or ignored entirely.

10.4 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.

10.4.1 Standard Data Section Names

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

Table 10-1. Traditional Section Names

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

The name “bss” dates back several decades, and means memory “Block Started by Symbol”. By convention, bss memory is filled with zeros during program start-up. The traditional section names are considered to have implied attributes as listed in the table above. The `code` section 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 5.1. Directives that Define Sections. For C applications, the XC32 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 C compiler guide relevant for your target device.

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 5.1. Directives that Define Sections.

10.4.2 Data Initialization Template

As noted in 10.4.1. 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 10-2. 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

10.4.3 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` for PIC32M devices or `libpic32c.a` for PIC32C/SAM devices.

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

Table 10-3. 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` linker 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 *`device family`*`/lib` 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.

10.5 Stack Allocation

The MPLAB XC32 C compiler dedicates general-purpose register 29 (PIC32M devices) or 13 (PIC32C/SAM devices) 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 C compiler user's guide relevant for your target device 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.

10.6 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.
Note: If no option is specified, the heap size defaults to 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.

10.7 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.

10.8 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.

Filename	Description
`devicename.ld`	Device linker script
`vector_offset_init.o`	Vector Offset initialization
`crt0_boot_isa.o`	Device runtime startup code

10.8.1 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))
 }
 }
```

10.8.2 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
```

10.8.3 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.

10.8.4 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.

For further information on these function attributes, see your compiler's user's guide.

11. Linker Examples

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 C compiler user's guide relevant for your target device. 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 5.5. 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.

11.1 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.

11.2 Locating Objects at a Specific Address

In this example, the C language 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
```

11.3 Locating Functions at a Specific Address

In this example, the C language function `func` is located at a specific address.

```
#include <stdio.h>
void __attribute__((address(0x9d002000))) func(void)
{ ... }
```

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:
               ......
```

11.4 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:
         ...
```

12. Linker Errors/Warnings

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.

12.1 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 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`

12.2 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.

12.2.1 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.

12.2.2 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.

12.2.3 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.

12.2.4 C

Cannot PROVIDE assignment to location counter.

An assignment to the special dot symbol can only be done during allocation. A *PROVIDE* command cannot use an assignment to the location counter. Remove the erroneous statement from the linker script.

Cannot 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’.

12.2.5 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.

12.2.6 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.

12.2.7 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.

12.2.8 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`.

12.2.9 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.

12.2.10 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.

12.2.11 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.

12.2.12 U

Undefined MEMORY region *region* referenced in expression.

The expression referenced a `MEMORY` region that does not exist in the linker script.

12.3 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.

12.3.1 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.

12.3.2 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.

12.3.3 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.

12.3.4 R

Redeclaration of memory region *name*.

The `MEMORY` region has been declared more than once in the linker script.

12.3.5 U

Undefined reference to *name*.

The symbol is undefined.

13. MPLAB XC32 Object Archiver/Librarian

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.

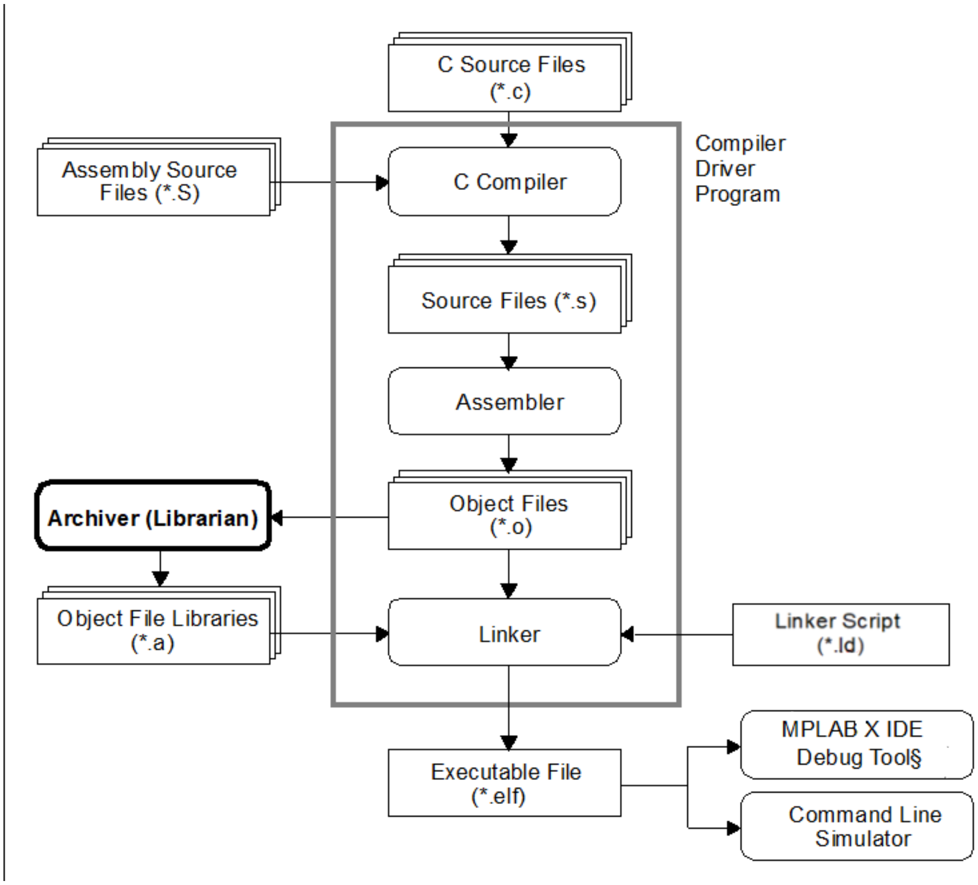
You may use `xc32-nm -s` or `xc32-nm --print-armap` to list this index table. If an archive lacks the table, another form of the 32-bit archiver/librarian called `xc32-ranlib` can be used to add only the table.

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.

13.1 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 the figure below for an overview of the tools process flow.

Figure 13-1. MPLAB X IDE Archiver Tools Process Flow



13.2 Feature Set

Notable features of the archiver include:

- Available for Linux, macOS, and Windows
- Command Line Interface

13.3 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.

13.4 Syntax

The general form of an archiver command looks like the following.

```
xc32-ar [-]P[MOD [RELPOS] [COUNT]] ARCHIVE [MEMBER...]
xc32-ar -M [ <mri-script ]
```

13.5 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 13-1. 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 13-2. 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.

...........continued

Option	Function
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.

13.6 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.
- The + character 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.

The following table shows the commands you can use in archiver scripts, or when using the archiver interactively. Three of them have special significance.

Table 13-3. Archiver Scripts Commands

Option	Function
`OPEN` or `CREATE`	Specify a “current archive,” which is a temporary file required for most of the other commands.
`SAVE`	Commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.

`ADDLIB` *`ARCHIVE`*
`ADDLIB` *`ARCHIVE`* `(`*`MODULE`*`,`
`MODULE``, . . .`*`MODULE`*`)`
	Add all the contents of *`ARCHIVE`* (or, if specified, each named *`MODULE`* from *`ARCHIVE`*) to the current archive.
	Requires prior use of `OPEN` or `CREATE`.

`ADDMOD` *`MEMBER`*`,` *`MEMBER`*`, ...` *`MEMBER`*	Add each named *`MEMBER`* as a module in the current archive. Requires prior use of `OPEN` or `CREATE`.
`CLEAR`	Discard the contents of the current archive, canceling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.
`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`.

...........continued

Option	Function
`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`.

14. Other Utilities

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 as shown in the following table.

Table 14-1. ADDITIONAL BINARY UTILITIES

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.

14.1 xc32-bin2hex Utility

The binary-to-hexadecimal (`xc32-bin2hex`) utility converts binary files (such as `.elf` files from the 32-bit linker) to Intel hex format files, suitable for loading into device programmers. The utility determines whether the binary file was built for a PIC32M or PIC32C/SAM device and will automatically run the appropriate internal utility to perform the conversion. Different options are permissible with each device family.

14.1.1 Input/Output Files

- Input: ELF formatted binary object files
- Output: Intel hex files

14.1.2 Syntax

Command line syntax is:

```
xc32-bin2hex [options] file
```

The following example converts the absolute ELF executable file `hello.elf` to a HEX file with the name `hello.hex`:

```
xc32-bin2hex hello.elf
```

14.1.3 Options

The following `xc32-bin2hex` options are supported.

Table 14-2. XC32-BIN2HEX Options

Option	Function
`-a` or `--sort`	Sort sections by address.
`-i` or `--virtual`	Use virtual addresses.
`-p` or `--physical`	Use physical addresses **(default)**.
`-v` or `--verbose`	Print verbose messages.
`-w` or `--write-sla`	Write the entry point into a linear start address (SLA) field as part of a type 5 record.
`-?` or `--help`	Print a help screen.
`-I` *`file`* or `--image=`*`file`*	Applicable for dual-core PIC32C devices capable of dual reset only, this option requests the generation of a secondary core image from the specified binary file. This image will be placed within `.c` and `.h` files.

...........continued

Option	Function
`-M` *`mode`* or `--image-copy-mode` *`mode`*	Applicable for dual-core PIC32C devices capable of dual reset only, this option indicates how the secondary core image (generated by the `--image` option) should be copied to SRAM. When *`mode`* is set to `auto`, the standard data-initialization template will copy the image to SRAM, using a simple compression algorithm on the payload. This is the default action if no option is specified. (See those sections pertaining to the Data-initialization Template in the *MPLAB® XC32 C/C++ Compiler User's Guide for PIC32C/SAM MCUs*.) When *`mode`* is set to `manual`, only the contents of the secondary core image are generated, and the end-user is responsible for copying the secondary image contents to its relevant memory.
`-C` *`name`* or `--image-generated-c` *`name`*	Applicable for dual-core PIC32C devices capable of dual reset only, this option specifies the base name (and optionally, a path) that will be used for the C source file containing the secondary core image (requested by the `--image` option). If the `--image-generated-h` option has not been specified, this option also supplies the basename for the generated `.h` header file.
`-H` *`name`* or `--image-generated-h` *`name`*	Applicable for dual-core PIC32C devices capable of dual reset only, this option specifies the base name (and optionally, a path) that will be used for the `.h` header file appropriate for the source file that contains the secondary core image (requested by the `--image` option).
`-A` *`offset`* or `--image-offset` *`offset`*	Applicable for dual-core PIC32C devices capable of dual reset only, this option specifies the boot memory address offset. The *`offset`* argument is a hexadecimal value.

Note: See the *PIC32MX Family Reference Manual* (DS61115) for a description of the PIC32MX Virtual-to-Physical Fixed Memory Mapping.

The following is example output when running verbose mode on an Elf file generated for a SAM device.

```
% xc32-bin2hex -v main.elf
Microchip Language Tool Shell Version 4.20 (Build date: Sep 16 2022).
Copyright (c) 2012-2017 Microchip Technology Inc. All rights reserved

Offsetting PFM addresses by 0x0000000000000000 and BFM address by 0x0000000000000000
writing main.hex

section                        byte address     length    (dec)
---------------------------------------------------------------
.vectors                           0x400000      0x168    (360)
.text                              0x400168       0x20    (32)
.text%8                            0x400188      0x150    (336)
.text.Reset_Handler%9              0x4002d8       0x9c    (156)
.dinit%10                          0x400374       0x70    (112)
.text.__pic32c_data_initialization_impl%11          0x4003e4       0x60    (96)
.text.__libc_start_init%12         0x400444       0x5c    (92)
.text%13                           0x4004a0       0x42    (66)
.text.Dummy_Handler%16             0x4004e2        0x2    (2)
.text%14                           0x4004e4       0x40    (64)
.text.__pic32c_data_initialization%15               0x400524        0x4    (4)
.text.__dummy%17                   0x400528        0x2    (2)

Total program memory used (bytes):               0x52a    (1322)
```

The following example shows a command line sequence that will generate secondary core image source files from the `main.elf` binary image file. This command will create `secCoreImage.c` and `secCoreImage.h` files.

```
xc32-bin2hex --image main.elf --image-offset 100 --image-generated-c secCoreImage main.elf
```

The generated C source file can be added to the application (core 0) project, where it can be used to initialize the secondary core (core 1) RAM. The header file must be placed in the preprocessor's search path specified for the application project.

14.2 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.

14.2.1 Input Files

The `xc32-nm` utility takes ELF object file as input. If no object files are listed as arguments, `xc32-nm` attempts to read from the file `a.out`.

14.2.2 Syntax

The general command-line usage of `xc32-nm` 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... ]
```

14.2.3 Options

The long and short forms of options, shown in the following table as alternatives, are equivalent.

Table 14-3. 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`.

...........continued

Option	Function
`-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.

14.2.4 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). The table below shows the symbol types:

Table 14-4. 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 a bss section).
`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.

14.3 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.

14.3.1 Input Files

The `xc32-objdump` utility takes ELF object files as input. If no object files are listed as arguments, `xc32-objdump` attempts to read from the file `a.out`.

14.3.2 Syntax

The general command-line usage of `xc32-objdump` 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...
```

The *`OBJFILE`*... are the object files to be examined. When you specify archives, `xc32-objdump` shows information on each of the member object files.

14.3.3 Options

The long and short forms of options, shown in the following table, 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 14-5. 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.

...........continued

Option	Function
-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.
-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.

14.4 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).

14.4.1 Input/Output Files

The xc32-ranlib utility takes ELF object file as input, storing the index into the same object file.

14.4.2 Syntax

The general command-line usage of `xc32-ranlib` is:

```
xc32-ranlib [-v | -V | --version] ARCHIVE
```

```
xc32-ranlib [-h | --help]
```

14.4.3 Options

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

Table 14-6. XC32-RANLIB Options

Option	Function

`-v`
`-V`
`--version`
	Show the version number of `xc32-ranlib`

`-h` `--help`	Print a help message

14.5 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.

14.5.1 Input/Output Files

The `xc32-size` utility takes ELF object or library archive files as input, printing the results to the standard output stream.

14.5.2 Syntax

The general command-line usage of `xc32-size` is:

```
xc32-size [ -A | -B | --format=compatibility ]
[ --help ]
[ -d | -o | -x | --radix=number]
[ -t | --totals ]
[ -V | --version ]
[objfile...]
```

14.5.3 Options

The `xc32-size` options are shown below.

Table 14-7. 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`.

...........continued

Option	Function
`-t` `--totals`	Show totals of all objects listed (Berkeley format listing mode only).
`-V` `--version`	Display the version number of `xc32-size`.

14.5.4 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
```

14.6 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.

14.6.1 Input/Output Files

The `xc32-strings` utility takes ELF object files as input, printing the results to the standard output stream.

14.6.2 Syntax

The general command-line usage of `xc32-strings` 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...
```

14.6.3 Options

The long and short forms of options, shown in the following table as alternatives, are equivalent.

Table 14-8. XC32-STRINGS Options

Option	Function

`-a`
`--all`
`-`
	Do not scan only the initialized and loaded sections of object files; scan the whole files.

...........continued

Option	Function
`-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.

14.7 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.

14.7.1 Input/Output Files

The `xc32-strip` utility takes ELF object or library archive files as input. If no files are listed as arguments, `xc32-strip` attempts to read from the file `a.out`. Output is written to the same object or library archive files.

14.7.2 Syntax

The general command-line usage of `xc32-strip` 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...
```

14.7.3 Options

The long and short forms of options, shown in the following table as alternatives, are equivalent.

Table 14-9. 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.

...........continued

Option	Function
`-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 a dot (`.`).

15. Useful Tables

15.1 ASCII Character Set

Table 15-1. ASCII Character Set

	Most Significant Character								
Least Significant Character	Hex	0	1	2	3	4	5	6	7
	0	NUL	DLE	Space	0	@	P	‘	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	’	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

15.2 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

16. Deprecated Features

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.

16.1 Assembler Directives that Define Sections

The following `.section` directive format was deprecated in XC32 v2.00. The new directive format may be found in 5.1. Directives that Define Sections.

16.1.1 Section Directive (Deprecated Form)

The `.section` *`name`* `[,` *`flags`*`] [, @`*`type`*`]` directive assembles the following code into a section named `name`. This form of the directive should not be used in new code. Use the alternate form, discussed in 5.1.5. Section Directive. 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 16-1. 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
```

17. GNU Free Documentation License

18. Document Revision History

Revision A (2013)

- Initial release of the document.

Revision B (May 2023)

- This guide has been migrated to a new authoring and publication system; you may see differences in the formatting compared to previous revisions
- Added new directives, options, and features pertaining to the PIC32C/SAM family of devices.

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **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 design partner 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

Product Change Notification Service

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

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Product Identification System

To order or obtain information, e.g., on pricing or delivery, refer to the factory or the listed sales office.

PART NO.	[X][1]	-	X	/XX	XXX
Device	Tape and Reel Option		Temperature Range	Package	Pattern

Device:	PIC16F18313, PIC16LF18313, PIC16F18323, PIC16LF18323	
Tape and Reel Option:	Blank	= Standard packaging (tube or tray)
	T	= Tape and Reel[1]
Temperature Range:	I	= -40°C to +85°C (Industrial)
	E	= -40°C to +125°C (Extended)
Package:[2]	JQ	= UQFN
	P	= PDIP
	ST	= TSSOP
	SL	= SOIC-14
	SN	= SOIC-8
	RF	= UDFN
Pattern:	QTP, SQTP, Code or Special Requirements (blank otherwise)	

Examples:

- PIC16LF18313- I/P Industrial temperature, PDIP package
- PIC16F18313- E/SS Extended temperature, SSOP package

Notes:

1. Tape and Reel identifier only appears in the catalog part number description. This identifier is used for ordering purposes and is not printed on the device package. Check with your Microchip Sales Office for package availability with the Tape and Reel option.
2. Small form-factor packaging options may be available. Please check www.microchip.com/packaging for small-form factor package availability, or contact your local Sales Office.

Microchip Devices Code Protection Feature

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

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods being used in attempts to breach the code protection features of the Microchip devices. We believe that these methods require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Attempts to breach these code protection features, most likely, cannot be accomplished without violating Microchip's intellectual property rights.
- Microchip is willing to work with any customer who is concerned about the integrity of its code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "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.

Legal Notice

Information contained in this publication is provided for the sole purpose of designing with and using Microchip products. Information regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AgileSwitch, APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, Inter-Chip Connectivity, JitterBlocker, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

ISBN: 978-1-6683-2479-0

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
www.microchip.com/support
Web Address:
www.microchip.com

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

Austin, TX
Tel: 512-257-3370

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

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

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

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

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

China - Beijing
Tel: 86-10-8569-7000

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

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

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

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

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

China - Shanghai
Tel: 86-21-3326-8000

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

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

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

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

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

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

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

India - Pune
Tel: 91-20-4121-0141

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

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

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

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

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

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

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

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

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

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

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

Denmark - Copenhagen
Tel: 45-4485-5910
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

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

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-72400

Germany - Karlsruhe
Tel: 49-721-625370

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

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra’anana
Tel: 972-9-744-7705

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

Italy - Padova
Tel: 39-049-7625286

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

Norway - Trondheim
Tel: 47-72884388

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

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

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820