MPLAB® XC16 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/.

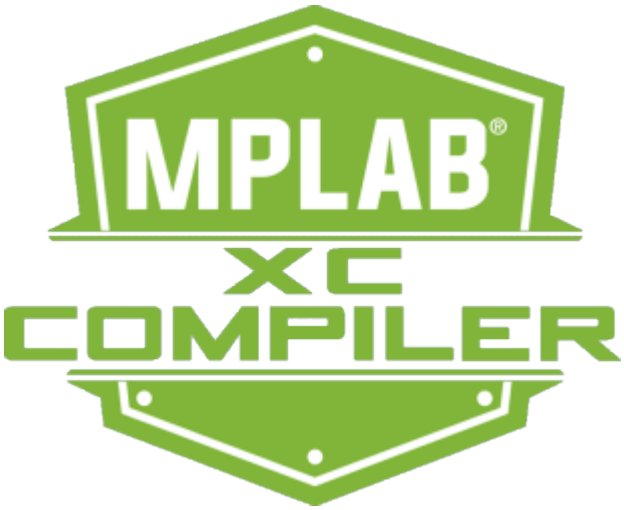

Table of Contents

1. Preface

MPLAB® XC16 Assembler and Linker documentation and support information is discussed in this section.

1.1 GNU Free Documentation License Notice

1.2 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® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
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:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`

.....continued		
Description	**Represents**	**Examples**
Italic Courier New	A variable argument	*file.o*, where *file* can be any valid filename
Square brackets [ ]	Optional arguments	`mcc18 [options] file [options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`

1.3 Recommended Reading

This guide describes how to use the MPLAB XC16 Assembler, Linker and utilities. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Release Notes (Readme Files)

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

MPLAB® XC16 C Compiler User's Guide (DS50002071)

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

16-Bit Language Tools Libraries (DS50001456)

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

Device-Specific Documentation

The Microchip website contains many documents that describe 16-bit device functions and features, including:

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

C Standards Information

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

This standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability and efficient execution of C language programs on a variety of computing systems.

C Reference Manuals

Harbison, Samuel P. and Steele, Guy L., *C A Reference Manual*, Fourth Edition, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, N.J. 07632.

Kochan, Steven G., *Programming In ANSI C*, Revised Edition. Hayden Books, Indianapolis, Indiana 46268.

Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Van Sickle, Ted., *Programming Microcontrollers in C*, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

2. Assembler Overview

MPLAB® XC16 Assembler produces relocatable machine code from symbolic assembly language for the dsPIC® DSC and PIC24 MCU families of devices. The assembler is an application that provides a platform for developing assembly language code. The assembler is a port of the GNU assembler from the Free Software Foundation.

2.1 Feature Set

Notable features of the assembler include:

- Support for the entire 16-bit instruction set
- Support for fixed-point and floating-point data
- Support for ELF and COFF object formats
- Available for Windows® OS, Linux® OS and macOS®
- Command Line Interface
- Rich Directive Set
- Flexible Macro Language

2.2 Assembler Usage

The MPLAB XC16 Assembler translates assembly source files into relocatable object files. These object files can then be put into an archive (MPLAB XC16 Object Archiver/Librarian) or linked with other relocatable object files and archives to create an executable file (MPLAB XC16 Object Linker). See the "*MPLAB XC16 C Compiler User's Guide*" (DS50002071) for an overview of the tools process flow.

Typically the command-line driver, `xc16-gcc`, is used to invoke the assembler as it can be passed assembler source files as input; however, the options for the assembler are supplied here for instances where the assembler is being called directly, or when options need to be set in the assembler tab assembler category of the Project Properties window.

The assembler command line may contain options and file names. For details on command line option syntax, see 3.1. Command-Line Syntax.

Note that the assembler will not produce any messages unless there are errors or warnings – there are no "assembly completed" messages. For more on messages, see 7. Assembler Errors/Warnings/Messages.

2.3 Input/Output Files

Standard assembler input and output files are listed below.

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

MPLAB XC16 Assembler does not generate error files, hex files, or symbol and debug files. The assembler is capable of creating a listing file and a relocatable object file (that may or may not contain debugging information). MPLAB XC16 Object Linker is used with the assembler to produce the final object files, map files and final executable file for debugging with MPLAB X IDE.

2.3.1 Source File

The assembler accepts, as input, a source file that consists of 16-bit device 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. See the "*MPLAB® XC16 C Compiler User's Guide*" (DS50002071) for more details on the C compiler driver.

For more information, see 3.1. Command-Line Syntax and 5. Assembler Directives.

Example 2-1. Sample Assembler Code

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

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

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

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

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

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

        .end
```

2.3.2 Object File

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

By default, the name of the object file created is `a.out`. Specify the -o option (see 3. Assembler Command Line Options) on the command line to override the default name.

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

```
xc16-as -omf=elf test.s
xc16-as -omf=coff test2.s
```

Alternatively, the environment variable XC16_OMF may be used to specify object file format for the 16-bit language tools.

2.3.3 Listing File

The assembler has the capability to produce listing files. For details on how to generate a listing file and the components of that file, see 6. Assembler Listing File.

3. Assembler Command Line Options

MPLAB XC16 Assembler may be used on the command line interface as well as with MPLAB X IDE. The following options may be used with either of these interfaces.

3.1 Command-Line Syntax

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

```
xc16-as [options|sourcefiles]...
```

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

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

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

  Note: Command line options are case sensitive.

3.2 Options that Modify the Listing Output

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

3.2.1 -a[suboption] [=file]

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

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

If no suboptions are specified, the default suboptions 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`. Most of the examples in the following sections combine the section's suboption with `-al`, because `-al` is required for an assembly listing.

3.2.1.1 -ac

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

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

Example 3-1. Listing File Generated with -al Command Line Option

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

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

Example 3-2. Listing File Generated with -alc Command Line Option

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

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

3.2.1.2 -ad

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

Example 3-3. Listing File Generated with -alhd Command Line Option

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

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

3.2.1.3 -ah

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

Example 3-4. Listing File Generated with -alh Command Line Option

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

   1                            .file "example2.4.c"
   2                            .text
   3                            .align  2
   4                            .def    _main
   5                            .val    _main
   6                            .scl    2
   7                            .type   044
   8                            .endef
   9                            .global _main  ; export
  10                     _main:
  11                            .def    .bf
  12                            .val    .
  13                            .scl    101
   1:example2.4.c **** extern int ADD (int, int);
   2:example2.4.c ****
   3:example2.4.c **** int
   4:example2.4.c **** main(void)
   5:example2.4.c **** {
  14                            .line   5
  15                            .endef
  16                            .set    _ ___PA_ ___,1
  17 000000  00 00 FA           lnk     #0
  18
```

```
  6:example2.4.c ****   return ADD(4, 5);
 19                             .ln     6
 20 000002  51 00 20            mov     #5,w1
 21 000004  40 00 20            mov     #4,w0
 22 000006  00 00 02            call    _ADD
 22         00 00 00
  7:example2.4.c **** }
 23                             .ln     7
 24                             .def    .ef
 25                             .val    .
 26                             .scl    101
 27                             .line   7
 28                             .endef
 29
 30 00000a  00 80 FA            ulnk
 31 00000c  00 00 06            return
 32                             .set    _ ___PA_ ___,0
 33                             .def    _main
 34                             .val    .
 35                             .scl    -1
 36                             .endef
 37
 38                             .end
```

3.2.1.4 -ai

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

Example 3-5. Listing File Generated with -ai Command Line Option

```
SECTION INFORMATION:

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

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

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

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

3.2.1.5 -al

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

3.2.1.6 -am

`-am` expands macros in a listing. The first example shows a listing where the `-am` option was not used. The second example shows a listing for the same source where the `-am` option was used.

Example 3-6. Listing File Generated with -al Command Line Option

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

   1                             .text
   2                             .macro div_s reg1, reg2
   3                               repeat #18-1
   4                               div.sw \reg1,\reg2
   5                             .endm
   6
   7                             .macro div_u reg1, reg2
   8                               repeat #18-1
   9                               div.uw \reg1,\reg2
```

```
  10                            .endm
  11
  12 000000  40 01 20           mov #20, w0
  13 000002  52 00 20           mov #5, w2
  14 000004  11 00 09           div_u w0, w2
  14         02 80 D8
  15
  16 000008  00 02 BE           mov.d w0, w4
  17
  18 00000a  40 01 20           mov #20, w0
  19 00000c  B3 FF 2F           mov #-5, w3
  20 00000e  11 00 09           div_s w0, w3
  20         03 00 D8
```

Example 3-7. Listing File Generated with -alm Command Line Option

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


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

Note: > signifies expanded macro instructions.

3.2.1.7 -an

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

Example 3-8. Listing File Generated with -al Command Line Option

```
MPLAB ASM30 Listing:  example2.7.s            page 1
User's Guide Example
  Listing Options
   1                            .text
   2                        .title "User's Guide Example"
   3                        .sbttl "  Listing Options"
   4                              .psize 10
   5
   6 000000  50 00 20           mov #5, w0
   7 000002  61 00 20           mov #6, w1
MPLAB ASM30 Listing:  example2.7.s            page 2
User's Guide Example
  Listing Options
   8 000004  01 01 40           add w0, w1, w2
   9                            .eject
MPLAB ASM30 Listing:  example2.7.s            page 3
```

```
User's Guide Example
  Listing Options
  10
  11 000006  24 00 20           mov #2, w4
  12 000008  03 00 09           repeat #3
  13 00000a  04 22 B8           mul.uu w4, w4, w4
  14
  15 00000c  16 00 20           mov #1, w6
  16 00000e  64 33 DD           sl w6, #4, w6
MPLAB ASM30 Listing:  example2.7.s            page 4
User's Guide Example
  Listing Options
  17
  18 000010  06 20 E1           cp w4, w6
  19 000012  00 00 32           bra z, done
  20
  21 000014  00 00 00           nop
  22
  23                    done:
MPLAB ASM30 Listing:  example2.7.s            page 5
User's Guide Example
  Listing Options
  24
  25                            .end
```

Example 3-9. Listing File Generated with -aln Command Line Option

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

3.2.1.8 -as

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

Example 3-10. Listing File Generated with -as Command Line Option

```
MPLAB ASM30 Listing:  sample2b.s

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

UNDEFINED SYMBOLS
```

```
_i
_j
```

3.2.1.9 -a=file

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

3.2.2 --listing-lhs-width

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

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

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

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

3.2.3 --listing-lhs-width2

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

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

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

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

3.2.4 --listing-rhs-width

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

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

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

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

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

3.2.5 --listing-cont-lines

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

using the `--listing-cont-lines` option. The text in bold shows the continuation lines used for the output data column of the listing.

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

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

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

3.3 Options that Control Informational Output

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

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

3.3.1 --fatal-warnings

Warnings are treated as if they were errors.

3.3.2 --no-warn (-W)

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

3.3.3 --warn

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

3.3.4 -J

No warnings are issued about signed overflow.

3.3.5 --help

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

3.3.6 --target-help

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

3.3.7 --version

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

3.3.8 --verbose (-v)

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

3.4 Options that Control Output File Creation

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

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

3.4.1 -g

Generate symbolic debugging information.

Note: For COFF, the option -g does not work with any section other than .text.

3.4.2 --keep-locals (-L)

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

3.4.3 -o objfile

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

3.4.4 -omf = format

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

3.4.5 -R

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

3.4.6 --relax

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

3.4.7 --no-relax

Turn relaxation off. This is the default behavior.

3.4.8 -Z

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

3.4.9 -MD file

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

3.5 Other Options

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

3.5.1 --defsym sym=value

Define symbol sym to given value.

3.5.2 -I dir

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

3.5.3 -p, --processor=PROC

Specify the target processor, e.g.:

```
xc16-as -p30F2010 file.s
```

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

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

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

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

Macros for the device families (see Section 3.5.6 "Predefined Symbols") are defined based on target processor setting.

4. MPLAB XC16 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:

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, this will be changed to the appropriate numeric value. If you have a syntax error that occurs at the single character, the assembler will not display '`b`', but instead display the first digit of the decimal equivalent.

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

The internal processor does not do:

1. Macro preprocessing.
2. Include file handling.
3. Anything else you may get from your C compiler's preprocessor.

You can do include file preprocessing with the `.include` directive. See 5. Assembler Directives.

You can use the C compiler driver to get other C-style preprocessing by giving the input file a `.S` suffix. See the "*MPLAB® XC16 C Compiler User's Guide*" (DS50002071) for more information.

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

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 Label

A label is one or more characters chosen from the set composed of all letters, digits, the underline character (_), and the period (.). Labels may not begin with a decimal digit, except for the special case of a local symbol. (See 4.5.2. Local Symbols for more information.) Case is significant. There is no length limit; all characters are significant.

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

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

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

4.2.2 Mnemonic

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

See the "*16-bit MCU and DSC Programmer's Reference Manual*" (DS70000157) for more details.

4.2.3 Directive

Assembler directives are commands that appear in the source code but are not translated directly into machine code. Directives are used to control the assembler, its input, output and data allocation. The first character of a directive is a dot (`.`). More details are provided in 5. Assembler Directiveson the available directives.

4.2.4 Operands

Each machine instruction takes 0 to 8 operands. See the "*16-bit MCU and DSC Programmer's Reference Manual*" (DS70000157). Operands provide data and addressing information to the instruction. Operands must be separated from mnemonics by one or more spaces or tabs.

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

4.2.4.1 Literals

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

Examples:

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

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

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

4.2.4.2 File Registers

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

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

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

4.2.4.3 Registers

The following register names are built into the assembler:

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

4.2.4.4 Condition Codes

Condition codes are used with `BRA` instructions. See the "*16-bit MCU and DSC Programmer's Reference Manual*" (DS70000157) for more details.

```
bra C, label
```

4.2.4.5 Destination Select

The PIC18CXXX-compatible instructions accept `WREG` as an optional argument to specify whether the result should be placed into `WREG` (W0) or into the file register. See the "*16-bit MCU and DSC Programmer's Reference Manual*" (DS70000157) for more details.

```
add sym, WREG
```

4.2.4.6 Accumulator Select

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

```
ADD A
```

4.2.5 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 are provided in 5. Assembler Directives on the available directives.

4.2.6 Comments

Comments can be represented in the assembler as single-line or multiple-line comments.

4.2.6.1 Single-Line Comment

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

Example:

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

4.2.6.2 Multiple-line Comment

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

/* ... */. Multiple-line comments cannot be nested.

Example:

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

4.3 Characters

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not mnemonics and reserved words. Tabs are treated as equivalent to spaces.

4.3.1 Delimiters

All numbers and identifiers must be delimited by white space, non-alphanumeric characters or the end of a line.

4.3.2 Special Characters

There are a few characters that are special in certain contexts. Within a macro body, the character & is used for token concatenation. To use the bitwise & operator within a macro body, escape it by using && instead. In a macro argument list, the angle brackets < and > are used to quote macro arguments.

Other special characters are described below.

Table 4-1. Special Characters and Usage

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

4.4 Constants

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

```
.byte 74, 0112, 0b01001010, 0x4A, 0x4a, 'J', '\J';All the same value
.ascii "Ring the bell\7";A string constant
.float 0f-3141592653589793238462643383279502884197l.693993751E-40
```

4.4.1 Numeric Constants

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

4.4.1.1 Integers

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

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

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

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

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

4.4.1.2 Floating-Point Numbers

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

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

– an 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.

4.4.1.3 Fixed-Point Numbers

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

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

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

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

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

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

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

4.4.2 Character Constants

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

4.4.2.1 Characters

A single character may be written as a single quote immediately followed by that character, or as a single quote immediately followed by that character and another single quote. As an example, either `'a` or `'a'`.

The assembler accepts escape characters to represent special control characters. As an example, `'\n'` represents a new-line character. All accepted escape characters are listed in the table below.

Table 4-2. Escape Characters

Escape Character	Description	Hex Value
`\a`	Bell (alert) character	07
`\b`	Backspace character	08
`\f`	Form-feed character	0C
`\n`	New-line character	0A
`\r`	Carriage return character	0D
`\t`	Horizontal tab character	09
`\v`	Vertical tab character	0B
`\\`	Backslash	5C
`\?`	Question mark character	3F
`\"`	Double quote character	22

.....continued		
Escape Character	**Description**	**Hex Value**
\digit digit digit	Octal character code. The numeric code is 3 octal digits.	
\x hex-digits	Hex character code. All trailing hex digits are combined. Either upper or lower case x works.	

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

4.5 Symbols

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

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

4.5.1 Reserved Names

The following symbol names (case-insensitive) are reserved for the assembler.

Do not use .equ, .equiv or .set (See 5. Assembler Directives) with these symbols.

Table 4-3. Symbol Names – Reserved

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

4.5.2 Local Symbols

Local symbols are used when temporary scope for a label is needed. There are ten local symbol names, which can be reused throughout the program. They may be referred to by using the names '0', '1', ..., '9'. To define a local symbol, write a label of the form 'N:' (where N represents any digit 0-9). To refer to the most recent previous definition of that symbol, write 'Nb', using the same digit as when you defined the label. To refer to the next definition of a local label, write 'Nf'. The 'b' stands for "backwards" and the 'f' stands for "forwards". There is no restriction on how to use these labels; however, at any point in assembly, no more than 10 backward local labels and10 forward local labels may be referred to.

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

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

Table 4-4. Symbol Parts

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

```
00000100 <print_string>:
100:   80 00 78        mov.w     w0, w1
00000102 <L1·1>:
102:   11 04 e0        cp0.b     [w1]
104:   03 00 32        bra       Z, . + 0x8
106:   31 40 78        mov.b     [w1++], w0
108:   02 00 07        rcall     . + 0x6
10a:   fb ff 37        bra       . + 0xFFFFFFF8
0000010c <L9·1>:
10c:   00 00 06        return
```

4.5.3 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign ‘=’, followed by an expression. This is equivalent to using the `.set` directive (see 5. Assembler Directives).

```
PSV = 4
```

4.5.4 The Special DOT Symbol

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

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

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

```
. = .+2
```

is the same as saying:

```
.org .+2
```

The symbol ‘$’ is accepted as a synonym for ‘`.`’.

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

4.5.5 Using Executable Symbols in a Data Context

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

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

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

In this example the `mov` instruction implies a data context. Because symbol `msg` is located in an executable section, an error is reported. Possibly the programmer was trying to derive a pointer for use with the PSV window. The special operators described in 4.8. Special Operators can be used whenever an executable symbol must be referenced in a data context:

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

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

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

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

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

4.5.6 Predefined Symbols

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

Table 4-5. Predefined Symbols

Symbol	Definition
Device Family Symbols	
`__C30COFF`	16-bit compiler COFF output
`__C30ELF`	16-bit compiler ELF output
`__dsPIC30F`	dsPIC30F target device family
`__dsPIC33F`	dsPIC33F target device family
`__dsPIC33E`	dsPIC33EP target device family
`__PIC24F`	PIC24FJ target device family
`__PIC24FK`	PIC24FK target device family
`__PIC24H`	PIC24H target device family
`__PIC24E`	PIC24EP target device family
`__MCHP16`	No target device family specified
Feature Symbols	
`__HAS_DSP`	Device has a DSP engine
`__HAS_EEDATA`	Device has EEDATA memory
`__HAS_DMA`	Device has DMA memory

.....continued	
Symbol	**Definition**
__HAS_DMAV2	Device has DMA v2 support
__HAS_CODEGUARD	Device has Codeguard™ Security
__HAS_PMP	Device has Parallel Master Port (PMP)
__HAS_PMPV2	Device has PMP v2 support
__HAS_PMP_ENHANCED	Device has Enhanced PMP
__HAS_EDS	Device has EDS
__HAS_5VOLTS	Device is a 5-volt device

4.6 Expressions

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

4.6.1 Empty Expressions

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

4.6.2 Integer Expressions

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

Integer expressions involving symbols in program memory are evaluated in Program Counter (PC) units. On the 16-bit device, the PC increments by 2 for each instruction word.

Branch After a Label

Branch to the next instruction after label `L` by specifying `L+2` as the destination.

```
bra L+2
```

4.7 Operators

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

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

4.7.1 Prefix Operators

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

Table 4-6. Prefix Operators

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

4.7.2 Infix Operators

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

Table 4-7. Infix Operators

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

4.8 Special Operators

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

Table 4-8. Special Operators [DD]

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

* You cannot use two special operators in an expression.
All = Support for all devices
24H = Support for PIC24H MCUs; 24E = Support for PI24EP MCUs
33F = Support for dsPIC33F DSCs; 33E = Support for dsPIC33EP DSCs

4.8.1 Accessing Data in Program Memory

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

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

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

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

- Table Read/Write Instructions
- PSV Data Window
- EDS Data Window

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

As explained in the "*16-bit MCU and DSC Programmer's Reference Manual*" (DS70000157), SFRs can be used to specify the full flash address. For a PSV address, use `PSVPAG` (or `DSRPAG` for devices with EDS). For a table read/write address, use `TBLPAG`. For an EDS address, use `DSRPAG`.

4.8.1.1 Table Read/Write Instructions

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

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

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

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

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

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

4.8.1.2 PSV Data Window

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

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

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

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

```
; Enable Program Space Visibility (Note 1)
bset.b        CORCONL, #PSV
; Set PSVPAG (Note 2) to the page that contains the fib_data array.
mov        #psvpage(fib_data), w0
mov        w0, _PSVPAG
; Make a pointer to fib_data in the PSV data window
mov        #psvoffset(fib_data), w0
; Load the first data value
mov        [w0++], w1
```

Note: Some devices do not need PSV to be enabled. Please check the data sheet for your device. For devices with EDS, use `DSRPAG`. Please check the data sheet for your device.

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

4.8.1.3 EDS Data Window

The `edspage()` and `edsoffset()` operators can be used with the EDS data window. The EDS data window replaces the PSV data window in certain device families. However, these operators are supported on all devices.

The `edspage()` operator may be applied to any symbol in any on-chip memory space. The operator returns a 10-bit page value. Unlike `psvpage()`, a value of zero is never returned.

The edsoffset() operator may be applied to any symbol in any on-chip memory space. The operator returns a 16-bit data space pointer. Unlike psvoffset(), the value of this pointer may fall anywhere in the data address space (0x0 to 0xFFFF).

Suppose that a table of data is located in any on-chip memory space. To access this table through the EDS data window, use the edspage() and edsoffset() operators as follows:

```
; set DSRPAG to the page that contains the glob_data array
mov    #edspage(glob_data),w0
mov    w0, _DSRPAG
; make a pointer to glob_data
mov    #edsoffset(glob_data),w0
; Load the first data value
mov    [w0++], w1
```

In order to access multiple items from a data table, you must ensure that the table does not cross a page boundary. To prevent this, specify the page section directive when the data table is defined. If additional constant storage is required, simply create additional tables, each with its own symbol, and repeat the code sequence above to load the DSRPAG register and derive a pointer.

4.8.2 Obtaining a Program Address of a Symbol or Constant

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

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

4.8.3 Obtaining a Handle to a Program Address

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

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

4.8.4 Obtaining the DMA Offset of a Symbol – PIC24H/dsPIC33F Devices Only [DD]

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

```
.section *,bss,dma
buf:  .space 256
.text
mov  #dmaoffset(buf), W0
```

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

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

4.8.5 Obtaining the DMA Offset of a Symbol – PIC24EP/dsPIC33EP Devices Only [DD]

The `dmaoffset()` and `dmapage()` operators can be used to obtain the offset of a symbol within DMA memory.

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

4.8.6 Obtaining the Size of a Specific Section

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

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

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

4.8.7 Obtaining the Starting Address of a Specific Section

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

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

4.8.8 Accessing Functions in Boot or Secure Segments

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

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

In order to support the separate linking of boot and secure application segments, access entry points may be defined. Access entry points provide a way to transfer control across segments to a function that may not be defined at link time. For more information about access entry points, see 5.5. Directives that Define Symbols and 11.12. Boot and Secure Segments.

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

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

5. Assembler Directives

Directives are assembler commands that appear in the source code but are not usually translated directly into opcodes. They are used to control the assembler: its input, output, and data allocation. All 16-bit directives are preceded by a dot '.'.

Notes:

1. Directives are not instructions (i.e., `movlw`, `btfss`, `goto`, etc.). For instruction set information, consult your device data sheet.
2. Directives that are supported, but deprecated, are listed in Appendix A. "Deprecated Features."

5.1 Directives that Define Sections

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

Section directives are described in the following sections.

5.1.1 bss

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

Note: You cannot reserve all of data memory for symbols; you will need room for the stack. See 11.8. Stack Allocation.

Example

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

5.1.2 data

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

Note: You cannot reserve all of data memory for symbols; you will need room for the stack. See 11.8. Stack Allocation

Example

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

The linker collects initial values for section .data (and other sections defined with the data attribute) and creates a data initialization template. This template can be processed during application start-up to transfer initial values into memory. For C applications, a library function is called for this purpose automatically. Assembly projects can utilize this library by linking with the libpic30 library. For more information, see the discussion of Run-Time Library Support in 11.6. Initialized Data.

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

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

Example

```
 ; define an external memory region
 .memory _memory1, size(8192), origin(0)
 ; allocate a section in external memory
 .section mem1_sec1,memory(_memory1)
 .global _mem1_array1
_mem1_array1:
 .skip 50
```

5.1.4 pushsection *name* [, *attr1*[,...,*attrn*]]

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

5.1.5 popsection

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

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

See 17.1.1. section name [, "flags"].

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

Assemble the following code into a section named `name`. If the character * is specified for `name`, the assembler will generate a unique name for the section based on the input file name in the format `filename.sc`*nn*, 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 5.1.7.3. Reserved Section Names with Implied Attributes [DD]. If the section name matches a reserved name, the implied attributes will be assigned to that section. If the section name is not recognized as a reserved name, the default attribute will be `data` (initialized storage in data memory).

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

If the first optional argument is quoted, it is taken as one or more flags that describe the section attributes. Quoted section flags are deprecated. (See 17. 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.

5.1.7.1 Attributes that Represent Section Types [DD]

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 That Represent Section Types

Attribute	Description	Support
`auxflash`	Executable code in auxiliary program memory	24EP/33EP (some)
`auxpsv`	Constant is in auxilliary program memory	24EP/33EP (some)
`bss`	Uninitialized storage in data memory	All
`code`	Executable code in program memory	All
`data`	Initialized storage in data memory	All

.....continued

Attribute	Description	Support
eedata	Non-volatile storage in data EEPROM	30/24FxxK
heap	Memory for dynamic allocation in C	All
memory	External or user-defined memory	All
packedflash	Use the upper byte of Flash via packed storage	All
persist	Persistent storage in data memory	All
psv	Constants in program memory	All
stack	Processor stack	All

All = Supported on all devices
24X = Supported on PIC24X MCUs, where X can be EP, FxxK, FJ, FV, HJ.

30 = Supported on dsPIC30F

33X = Supported on dsPIC33X DSCs, where X can be EP, EV, FJ.

5.1.7.2 Attributes that Modify Section Types [DD]

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

Table 5-2. Attributes That Modify Section Types

Attribute*	Description	Attribute applies to**
address(a)	locate at absolute address a	auxflash, bss, code, data, eedata, heap, memory, persist, psv, stack
near	locate in the first 8K of memory	bss, data, persist
xmemory	locate in X address space	bss, data, heap, persist
ymemory	locate in Y address space	bss, data, heap, persist
reverse(n)	align the ending address +1	auxflash, bss, data, eedata, memory, persist, psv
align(n)	align the starting address	auxflash, bss, code, data, eedata, heap, memory, persist, psv, stack
noload	allocate, do not load	auxflash, bss, code, data, eedata, memory, persist, psv
merge(n)	mergable elements of size n***	auxflash, code, data, eedata, psv
info	do not allocate or load	auxflash, bss, code, data
dma	locate in DMA space	bss, data, persist
boot	locate in boot segment	bss, code, eedata, psv
secure	locate in secure segment	bss, code, eedata, psv
eds	locate in extended data space	bss, data, persist
shared	use section outside of application	auxflash, bss, code, data, eedata, memory, psv, stack
preserved	preserve variables on restart	bss, data, memory, persist
update	initialize variables on restart	bss, data, memory, persist
priority(n)	group variable initializations together.	auxflash, bss, code, data

.....continued		
Attribute*	**Description**	**Attribute applies to****
page	do not cross page boundary	auxflash, bss, code, data, persist, psv

* = Not all devices support all attributes.
** = See 5.1.7.1. Attributes that Represent Section Types [DD] for descriptions and device support.

*** = This attribute could be used by a linker to merge identical constants across input files.
If n=0, the section contains null-terminated strings of variable length.

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

Table 5-3. Combining Attributes that Modify Section Types

Attribute*	Attribute can be combined with*
address	near, xmemory, ymemory, noload, dma, boot, secure, eds, page
near	address, xmemory, ymemory, reverse, align, noload, merge
xmemory	address, near, reverse, align, noload, merge, eds, page
ymemory	address, near(30,33), reverse, align, noload, merge, eds, page
reverse	near, xmemory, ymemory(30/33), noload, merge, dma, boot, secure, eds, page
align	near, xmemory, ymemory(30/33), noload, merge, dma, boot, secure, eds, page
noload	address, near, xmemory, ymemory, reverse, align, dma, boot, secure, eds, page
merge	near, xmemory, ymemory, reverse, align, eds, page
info	N/A
dma	address, reverse, align, noload, eds, page
boot	address, reverse, align, noload, eds, page
secure	address, reverse, align, noload, eds, page
eds	address, xmemory, ymemory, reverse, align, noload, merge, dma, boot, secure, page
shared	address, near, xmemory, ymemory, reverse, align, noload, merge, dma, boot, secure, eds, preserved, update, priority, page
preserved	address, near, xmemory, ymemory, reverse, align, noload, merge, dma, eds, shared, priority, page
update	address, near, xmemory, ymemory, reverse, align, noload, merge, dma, eds, shared, priority, page
priority(n)	address, near, xmemory, ymemory, reverse, align, dma, boot, secure, eds, shared, preserved, update, page
page	address, xmemory, ymemory(30, 33), reverse, align, noload, merge, dma, boot, secure, eds

* = Not all devices support all attributes.

5.1.7.3 Reserved Section Names with Implied Attributes [DD]

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

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

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

33x = Supported on dsPIC33x DSCs

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

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

5.1.7.4 Section Directive Examples

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

5.1.7.5 Section Directive Examples - Boot/Secure Segments

Program Memory

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

```
 .section *,code,boot
 .global func1
func1:
 return
 .section *,code,secure
 .global func2
func2:
 return
```

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

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

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

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

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

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

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

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

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

Data Memory

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

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

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

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

Constants in Non-Volatile Memory

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

```
 .section *,psv,boot
 .global key1
key1:
 .ascii “abcdefg”
 .section *,eedata,boot
 .global key2
key2:
 .ascii “hijklm”
```

5.1.8 text

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

Example

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

5.2 Directives that Fill Program Memory

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

Fill directives are described below.

5.2.1 fillupper [*value*]

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

See the 5.2.4. Section Example table.

5.2.2 fillvalue [*value*]

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

See the Section Example table.

5.2.3 pfillvalue [*value*]

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

See the Section Example table.

5.2.4 Section Example

Memory			Instruction
			`.section .myconst, code`
			`.fillvalue 0x12`
			`.fillupper 0x34`
			`.pfillvalue 0x56`

.....continued			
Memory			Instruction
0x12	0x12	0x34	.fill 4
0x12	0x12		
		0x34	.align 2 ;Align to next p-word
0x56	0x56	0x56	.pfill 8
0x56	0x56	0x56	
0x56	0x56		
		0x56	.palign 2 ;Align to next p-word
			.fillvalue ;Reset fillvalue
			.pfillvalue ;Reset pfillvalue
0x00	0x00	0x34	.fill 4
0x00	0x00		
		0x34	.align 2 ;Align to next p-word
0x00	0x00	0x00	.pfill 8
0x00	0x00	0x00	
0x00	0x00		
		0x00	.palign 2 ;Align to next p-word

5.3 Directives that Initialize Constants

Constant initialization directives are listed below.

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

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

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

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

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

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

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

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

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

5.3.3 pascii "*string$_1$*"

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

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

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

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

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

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

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

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

5.3.6 pasciz "*string$_2$*"

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

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

Assembles one or more successive bytes in the current section.

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

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

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

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

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

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

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

Floating point numbers are in IEEE format (see Section 3.4.1.2 "Floating-Point Numbers.").

The following statements are equivalent:

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

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

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

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

Assembles one or more 2-byte fixed-point constants (range -1.0 <= f < 1.0) into consecutive addresses in little-endian format. Fixed-point numbers are in Q-15 format (see 4.4.1.3. Fixed-Point Numbers).

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

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

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

Floating point numbers are in IEEE format (see 4.4.1.2. Floating-Point Numbers).

The following statements are equivalent:

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

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

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

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

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

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

Floating point numbers are in IEEE format.

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

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

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

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

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

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

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

Same as 5.3.20. word expr1[, ..., exprn].

5.3.17 string "*str*"

Same as 5.3.4. asciz "string1" | <##>1 [, ..., "stringn" | <##>n].

5.3.18 pstring "*str*"

Same as 5.3.5. pasciz "string1" | <##>1 [, ..., "stringn" | <##>n].

5.3.19 pstring "*string*$_2$"

Same as 5.3.6. pasciz "string2".

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

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

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

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

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

5.4 Directives that Declare Symbols

Declare symbol directives are listed below.

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

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

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

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

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

5.4.3 extern *symbol*

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

5.4.4 global *symbol*
.globl *symbol*

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

5.4.5 lcomm *symbol*, *length*

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

5.4.6 weak *symbol*

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

5.5 Directives that Define Symbols

Define symbol directives are described below.

5.5.1 equ *symbol*, *expression*

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

5.5.2 equiv *symbol*, *expression*

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

5.5.3 set *symbol*, *expression*

Same as `.equ`.

5.6 Directives that Modify Section Alignment

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

5.6.1 Implicit Alignment in Program Memory

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

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

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

The following conditions cause implicit alignment in program memory:

1. Labels must be aligned to a valid address. For example, see the following source code:

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

   This generates implicit alignment as shown in the disassembly of section `.text`:

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

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

2. Instructions must be aligned to a valid address. For example, see the following source code:

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

   This generates implicit alignment as shown in the disassembly of section `.text`:

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

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

3. Transitions between p-type data directives (`.pbyte`, `.pspace`, etc). and normal data directives (`.byte`, `.space`, etc.), in either direction, are aligned to a valid address. For example, see the following source code:

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

This generates implicit alignment as shown in the disassembly of section `.text`:

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

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

5.6.2 Explicit Section Alignment Directives

Directives that explicitly modify section alignment are listed below.

5.6.2.1 align *algn*[, *fill*[, *max-skip*]]

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

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

fill is optional. If not specified:

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

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

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

5.6.2.2 palign *algn*[, *fill*[, *max-skip*]]

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

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

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

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

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

5.6.2.3 fill *repeat*[, *size*[, *fill*]]

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

size is optional and defaults to one, if omitted.

fill is optional. If not specified:

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

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

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

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

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

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

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

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

Note: A location counter is not an absolute address but the offset from the start of the section in which the `.org` occurs.

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

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

fill is optional. If not specified:

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

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

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

Note: A location counter is not an absolute address but the offset from the start of the section in which the `.porg` occurs.

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

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

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

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

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

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

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

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

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

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

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

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

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

5.6.2.9 struct *expression*

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

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

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

5.7 Directives that Format the Output Listing

Output listing format directives are described below.

5.7.1 eject

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

5.7.2 list

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

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

5.7.3 nolist

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

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

5.7.4 psize *lines*[, *columns*]

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

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

5.7.5 sbttl "*subheading*"

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

5.7.6 title "*heading*"

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

5.8 Directives that Control Conditional Assembly

Conditional assembly directives are described below.

5.8.1 else

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

5.8.2 elseif *expr*

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

5.8.3 endif

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

5.8.4 err

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

5.8.5 error "*string*"

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

5.8.6 if *expr*

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

5.8.7 ifdecl *symbol*

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

5.8.8 ifndecl *symbol*
.ifnotdecl *symbol*

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

5.8.9 ifdef symbol

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

5.8.10 ifndef *symbol*
.ifnotdef *symbol*

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

5.9 Directives for Substitution/Expansion

Substitution/expansion directives are described below.

5.9.1 exitm

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

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

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

For example, assembling:

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

is equivalent to assembling:

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

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

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

For example, assembling:

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

is equivalent to assembling:

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

5.9.4 macro *symbol arg*$_1$[=*default*] [, ..., *arg*$_n$ [=*default*]] ... .endm

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

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

For example, assembling:

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

```
.endm
display_int result
```

is equivalent to assembling:

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

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

```
 .macro LCONST name,value
 .equ \name,\value
 .equ &name&LO,(\value) & 0xFFFF
 .equ &name&HI,((\value)>>16) & 0xFFFF
 .endm
 LCONST seconds_per_day 60*60*24
 mov #seconds_per_dayLO,w0
 mov #seconds_per_dayHI,w1
```

`xc16-as` maintains a counter of how many macros have been executed in the psuedo-variable `\@`. This value can be copied to the assembly output, but only within a macro definition. In the following example, a recursive macro is used to allocate an arbitrary number of labeled buffers.

```
 .macro make_buffers num,size
BUF\@: .space \size
 .if (\num - 1)
 make_buffers (\num - 1),\size
 .endif
 .endm
 .bss
 make_buffers 4,16 ; create BUF0..BUF3, 16 bytes each
```

5.9.5 purgem "*name*"

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

5.9.6 rept *count*

...

.endr

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

For example, assembling

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

is equivalent to assembling

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

5.10 Miscellaneous Directives

Miscellaneous directives are described below.

5.10.1 abort

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

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

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

5.10.3 end

End program

5.10.4 fail *expression*

Generates an error or a warning. If the value of the expression is 500 or more, as will print a warning message. If the value is less than 500, `xc16-as` will print an error message. The message will include the value of expression. This can occasionally be useful inside complex nested macros or conditional assembly.

5.10.5 ident "*comment*"

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

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

The `.incbin` directive includes *`file`* verbatim at the current location. The file is assumed to contain binary data. The search paths used can be specified with the `-I` command-line option (see 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.

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

5.10.7 include "*file*"

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

5.10.8 loc *file-number*, *line-number*

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

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

The `.pincbin` directive includes *`file`* verbatim at the current location. The file is assumed to contain binary data. The search paths used can be specified with the `-I` command-line option (see 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 `.pincbin` directive.

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

5.10.10 print "*string*"

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

5.10.11 version "*string*"

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

5.11 Directives for Debug Information

Debug information directives are described below.

5.11.1 def *name*

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

5.11.2 dim

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

5.11.3 endef

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

5.11.4 file "*string*"

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

5.11.5 line *line-number*

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

5.11.6 scl *class*

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

5.11.7 size *expression*

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

5.11.8 size *name*, *expression*

Generated by compilers to include auxiliary information for debugging. This variation of `.size` is supported when the output file format is in Executable and Linking Format (ELF).

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

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

5.11.10 tag *structname*

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

5.11.11 type *value*

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

5.11.12 type *name*, *description*

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

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

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

5.11.13 uleb128 *expr$_1$*[,...,*expr$_n$*]

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

5.11.14 val *addr*

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

6. Assembler Listing File

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

6.1 Generation

To generate a listing file in MPLAB X IDE or on the command line, you will need to specify an option described in 3.2. Options that Modify the Listing Output. By default, a listing file is written to a `.lst` file.

6.2 Contents

The listing files produced by the assembler are composed of the several elements. The example below shows a sample listing file.

The example listing file contains these elements:

- **Header** – contains the name of the assembler, the name of the file being assembled, and a page number. This is not shown if the `-an` option is specified.
- **Title Line** – contains the title specified by the `.title` directive. This is not shown if the `-an` option is specified.
- **Subtitle** – contains the subtitle specified by the `.sbttl` directive. This is not shown if the `-an` option is specified.
- **High-level source** if the `-ah` option is given to the assembler. The format for high-level source is:
  `<line #>:<filename> **** <source>`

  For example:

  `1:hello.c **** #include <stdio.h>`
- **Assembler source** if the `-al` option is given to the assembler. The format for assembler source is:
  `<line #> <addr> <encoded bytes> <source>`

  For example:

  `245 000004 00 0F 78 mov w0,[w14]`

  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.
- **Symbol table** if the `-as` option is given to the assembler. Both, a list of defined and undefined symbols will be given. The defined symbols will have the format:

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

  For example:

  ```
  DEFINED SYMBOLS
  foo.s:229 .text:00000000 _main
  ```

  The undefined symbols will have the format:

  ```
  UNDEFINED SYMBOLS
  <symbol>
  ```

  For example:

  ```
  UNDEFINED SYMBOLS
  printf
  ```

Sample Assembler Listing File

```
MPLAB XC16_ASM_ Listing:  example1.1.s                      page 1
 Sample dsPIC Assembler Source Code
 For illustration only.
   1
   2                            .title " Sample dsPIC Assembler Source Code"
   3                            .sbttl " For illustration only."
   4
   5                            ; dsPIC registers
   6                            .equ CORCONL, CORCON
   7                            .equ PSV,2
   8
   9                            .section .const,psv
  10                     hello:
  11 0000 48 65 6C 6C           .ascii "Hello world!\n\0"
  11      6F 20 77 6F
  11      72 6C 64 21
  11      0A 00
  12
  13                            .text
  14                            .global _ ___reset
  15                     _ ___reset:
  16                            ; set PSVPAG to page that contains 'hello'
  17 000000  00 00 20           mov     #psvpage(hello),w0
  18 000002  00 00 88           mov     w0,PSVPAG
  19
  20                            ; enable Program Space Visibility
  21 000004  00 40 A8           bset.b  CORCONL,#PSV
  22
  23                            ; make a pointer to 'hello'
  24 000006  00 00 20           mov     #psvoffset(hello),w0
  25
  26                            .end


MPLAB XC16_ASM_ Listing:  example1.1.s                      page 2
 Sample dsPIC Assembler Source Code
 For illustration only.
DEFINED SYMBOLS
                             *ABS*:00000000 fake
        example1.1.s:10      .const:00000000 hello
        example1.1.s:15      .text:00000000 _ ___reset
                             .text:00000000 .text
                             .data:00000000 .data
                              .bss:00000000 .bss
                            .const:00000000 .const

UNDEFINED SYMBOLS
CORCON
PSVPAG
```

7. Assembler Errors/Warnings/Messages

MPLAB XC16 Assembler may generate errors, warnings and messages. To select the messages that are generated, see 3.3. Options that Control Informational Output.

For information on assembler limitations and known problems, see the compiler release notes (readme file).

7.1 Fatal Errors

The following errors indicate that an internal error has occurred in the assembler. Please contact Microchip TTechnical Support (support.microchip.com) if any of the following errors are generated:

- A dummy instruction cannot be used!
- bad floating-point constant: exponent overflow, probably assembling junk
- bad floating-point constant: unknown error code=error_code
- C_EFCN symbol out of scope
- Can’t continue
- Can’t extend frag num. chars
- Can’t open a bfd on stdout name
- Case value val unexpected at line _line_ of file “_file_”
- emulations not handled in this configuration
- error constructing pop_table_name pseudo-op table: err_txt
- expr.c(operand): bad atof_generic return val val
- failed sanity check.
- filename:line_num: bad return from bfd_install_relocation: val
- filename:line_num: bad return from bfd_install_relocation
- Inserting “name” into symbol table failed: error_string
- pic30_get_g_or_h_mode_value called with an invalid operand type
- pic30_get_p_or_q_mode_value called with an invalid operand type
- pic30_insert_dsp_writeback called with an invalid operand type
- pic30_insert_dsp_x_prefetch_operation called with an invalid offset
- pic30_insert_dsp_x_prefetch_operation called with an invalid operand type
- pic30_insert_dsp_y_prefetch_operation called with an invalid offset
- pic30_insert_dsp_y_prefetch_operation called with an invalid operand type
- invalid segment “name”; segment “name” assumed
- label “temp$” redefined
- macros nested too deeply
- missing emulation mode name
- multiple emulation names specified
- Relocation type not supported by object file format
- reloc type not supported by object file format
- rva not supported
- rva without symbol
- unrecognized emulation name ‘em’
- Unsupported BFD relocation size in bytes

7.2 Errors

Symbol
.abort detected. Abandoning ship.

User error invoked with the `.abort` directive.

.else without matching .if - ignored.

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

“.elseif” after “.else” - ignored

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” - ignored.

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

“.endif” without “.if”

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

.err encountered.

A user error invoked with the `.err` directive.

 # sign not valid in data allocation directive.

The # sign cannot be used within a data allocation directive (`.byte`, `.word`, `.pword`, `.long`, etc.)

 # warnings, treating warnings as errors.

The `--fatal-warnings` command line option was specified on the command line and warnings existed.

A

absolute address can not be specified for section '.const'.

Section `.const` is a C compiler resource. Although it is permissible for an application to allocate constants in section `.const` explicitly, it is not permissible to assign an absolute address for this section.

Absolute address must be greater than or equal to 0.

A negative absolute address was specified as the target for the DO or BRA instruction. The assembler does not know anything about negative addresses.

Alignment in CODE section must be at least 2 units.

The alignment value for the `.align` directive must be at least 2 units. Either no alignment was specified or an alignment less than 2 was specified. Modify the `.align` directive to have an alignment of at least 2.

Attributes for section 'name' conflict with implied attributes.

Certain section names have implied attributes. In this case, the attributes specified in a .section directive conflict with its implied attributes. See 5.1. Directives that Define Sections for more information.

B

backw. ref to unknown label “#:”, 0 assumed.

A backwards reference was made to a local label that was not seen. See 4.5.1. Reserved Names for more information on local labels.

bad defsym; format is --defsym name=value.

The format for the command line option `--defsym` is incorrect. Most likely, you are missing the = between the name and the value.

Bad expression.

The assembler did not recognize the expression. See 3. Assembler Command Line Options and 4. MPLAB XC16 Assembly Language for more details on assembler syntax.

bignum invalid; zero assumed.

The big number specified in the expression is not valid.

Byte operations expect an offset between -512 and 511.

The offset specified in [Wn+offset] or [Wn-offset] exceeded the maximum or minimum value allowed for byte instructions.

C

Cannot call a symbol (name) that is not located in an executable section.

Attempted to CALL a symbol that is not located in a CODE section.

Cannot create floating-point number.

Could not create a floating-point number because of exponent overflow or because of a floating-point exception that prohibits the assembler from encoding the floating-point number.

Cannot redefine executable symbol ‘s’.

A statement label or an executable section cannot be redefined with a `.set` or `.equ` directive.

Cannot reference executable symbol (name) in a data context.

An attempt was made to use a symbol in an executable section as a data address. To reference an executable symbol in a data context, the `psvoffset()` or `tbloffset()` operator is required.

Cannot use a constant as the argument of dmaoffset.

An attempt was made to use a constant as the argument to a dmaoffset.

Can not use dmaoffset on a symbol (name) that is not located in a dma section.

For some devices, the `dmaoffset()` operator can only be used on symbols that are located in dma memory.

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

The following operators can be applied to symbols in executable or read-only sections only: `tbloffset()`, `psvoffset()`, `tblpage()`, `psvpage()`, `handle()`, `paddr()`.

Cannot use operator on a symbol (name) that is not located in a code, psv or eedata section.

You cannot use one of the special operators (`tbloffset`, `tblpage`, `psvoffset`, `psvpage`, `handle` or `paddr`) on a symbol that is not located in a code, psv or eedata section.

Cannot use operator with this directive.

An attempt was made to use a special operator (`tbloffset`, `tblpage`, `psvoffset`, `psvpage`, `handle` or `paddr`) with a data allocation directive that does not allocate enough bytes to store the requested data.

Cannot write to output file.

For some reason, the output file could not be written to. Ensure that you have write permission to the file and that there is enough disk space.

Can’t open file_name for reading.

The specified input source file could not be opened. Ensure that the file exists and that you have permission to access the file.

D

directive directive not supported in pic30 target.

The pic30 target does not support this directive. This directive is available in other versions of the assembler, but the pic30 target does not support it for one reason or another. Please check 5. Assembler Directives for a complete list of supported directives.

duplicate “else” - ignored.

Two `.else` directives were specified for the same `.if` directive.

E

end of file inside conditional.

The file ends without terminating the current conditional. Add a `.endif` to your code.

end of macro inside conditional.

A conditional is unterminated inside a macro. The `.endif` directive to end the current conditional was not specified before seeing the `.endm` directive.

Expected comma after symbol-name: rest of line ignored.

Missing comma from the `.comm` directive after the symbol name.

Expected constant expression for fill argument.

The fill argument for the `.fill`, `.pfill`, `.skip`, `.pskip`, `.space` or `.pspace` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expected constant expression for new-lc argument.

The new location counter argument for the `.org` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expected constant expression for repeat argument.

The repeat argument for the `.fill`, `.pfill`, `.skip`, `.pskip`, `.space` or `.pspace` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expected constant expression for size argument.

The size argument for the `.fill` or `.pfill` directive must be a constant value. Attempted to use a symbol. Replace symbol with a constant value.

Expression too complex.

An expression is too complex for the assembler to process.

F

floating point number invalid; zero assumed.

The floating-point number specified in the expression is not valid.

I

Ignoring attempt to re-define symbol ‘symbol’.

The symbol that you are attempting to define with `.comm` or `.lcomm` has already been defined and is not a common symbol.

Invalid expression (expr) contained inside of the brackets.

Assembler did not recognize the expression between the brackets.

invalid identifier for “.ifdef”.

The identifier specified after the `.ifdef` must be a symbol. See 4.5.1. Reserved Names and 5.8. Directives that Control Conditional Assembly for more details.

Invalid mnemonic: ‘token’.

The token being parsed is not a valid mnemonic for the instruction set.

invalid listing option ‘optarg’.

The sub-option specified is not valid. Acceptable suboptions are `c`, `d`, `h`, `l`, `m`, `n`, `v` and `=`.

Invalid operands specified (‘insn’). Check operand #n.

The operands specified were invalid. The assembler was able to match n-1 operands successfully. Although there is no assurance that operand #n is the culprit, it is a general idea of where you should begin looking.

Invalid operand syntax (‘insn’).

This message usually comes hand-in-hand with one of the previous operand syntax errors.

Invalid post increment value. Must be +/- 2, 4 or 6.

Assembler saw [Wn]+=value, where value is expected to be a +/- 2, 4 or 6. value was not correct. Specify a value of +/- 2, 4 or 6.

Invalid post decrement value. Must be +/- 2, 4 or 6.

Assembler saw [Wn]-=value, where value is expected to be a +/- 2, 4 or 6. value was not correct. Specify a value of +/- 2, 4 or 6.

Invalid register in operand expression.

Assembler was attempting to find either pre- or post-increment or decrement. The operand did not contain a register. Specify one of the registers w0-w16 or W0-W16.

Invalid register in expression reg.

Assembler saw [junk] or [junk]+=n or [junk]-=n. Was expecting a register between the brackets. Specify one of the registers w0-w16 or W0-W16 between the brackets.

Invalid use of ++ in operand expression.

Assembler was attempting to find either pre- or post-increment. The operand specified was neither pre-increment [++Wn] nor post-increment [Wn++]. Make sure that you are not using the old syntax of [Wn]++.

Invalid use of -- in operand expression.

Assembler was attempting to find either pre- or post-decrement. The operand specified was neither pre-decrement [--Wn] nor post-decrement [Wn--]. Make sure that you are not using the old syntax of [Wn]--.

Invalid value (#) for relocation name.

The final value of the relocation is not a valid value for the operand associated with the given relocation.

'name' is not a valid attribute name.

While processing a .section directive, the assembler found an identifier that is not a valid section attribute.

L
Length of .comm “sym” is already #. Not changed to #.

An attempt was made to redefine the length of a common symbol.

M
misplaced)

Missing parenthesis when expanding a macro. The syntax `\(...)` will literally substitute the text between the parenthesis into the macro. The trailing parenthesis was missing from this syntax.

Missing model parameter.

Missing symbol in the `.irp` or `.irpc` directive.

Missing right bracket.

The assembler did not see the terminating bracket ‘]’.

Missing size expression.

The `.lcomm` directive is missing the length expression.

Missing ‘)’ after formals.

Missing trailing parenthesis when listing the macro formals inside of parenthesis.

Missing ‘)’ assumed.

Expected a terminating parenthesis ‘)’ while parsing the expression. Did not see one where expected so assumes where you wanted the trailing parenthesis.

Missing ‘]’ assumed.

Expected a terminating brace ‘]’ while parsing the expression. Did not see one where expected so assumes where you wanted the trailing brace.

Mnemonic not found.

The assembler was expecting to parse an instruction and could not find a mnemonic.

N

Negative of non-absolute symbol name.

Attempted to take the negative of a symbol name that is non-absolute. For example, `.word -sym`, where *sym* is external.

New line in title.

The `.title` heading is missing a terminating quote.

non-constant expression in “.elseif” statement.

The argument of the `.elseif` directive must be a constant value able to be resolved on the first pass of the directive. Ensure that any `.equ` of a symbol used in this argument is located before the directive. See 5.8. Directives that Control Conditional Assembly for more details.

non-constant expression in “.if” statement.

The argument of the `.if` directive must be a constant value able to be resolved on the first pass of the directive. Ensure that any `.equ` of a symbol used in this argument is located before the directive. See 5.8. Directives that Control Conditional Assembly for more details.

Number of operands exceeds maximum number of 8.

Too many operands were specified in the instruction. The largest number of operands accepted by any of the 16-bit device instructions is 8.

O

Only support plus register displacement (i.e., [Wb+Wn]).

Assembler found [Wb-Wn]. The syntax only supports a plus register displacement.

Operands share encoding bits. The operands must encode identically.

Two operands are register with displacement addressing mode [Wb+Wn]. The two operands share encoding bits so the Wn portion must match or be able to be switched to match the Wb of the other operand.

operation combines symbols in different segments.

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

operator modifier must be preceded by a #.

The modifier (`tbloffset`, `tblpage`, `psvoffset`, `psvpage`, `handle`) was specified inside of an instruction, but was not preceded by a #. Include the # to represent that this is a literal.

P

paddr modifier not allowed in instruction.

The `paddr` operator was specified in an instruction. This operator can only be specified in a `.pword` or `.long` directive as those are the only two locations that are wide enough to store all 24 bits of the program address.

PC relative expression is not a valid GOTO target.

The assembler does not support expressions which modify the PC of a GOTO destination such as “. + 4” or “sym + 100”.

R

Register expected as first operand of expression expr.

Assembler found [junk+anything] or [junk-anything]. The only valid expression contained in brackets with a + or a - requires that the first operand be a register.

Register or constant literal expected as second operand of expression expr.

Assembler found [Wn+junk] or [Wn-junk]. The only valid operand for this format is a register with plus or minus literal offset or a register with displacement.

Requested alignment 'n' is greater than alignment of absolute section 'name'

When the `address()` attribute is used to specify an absolute address for a section, it constrains the ability of the assembler to align objects within the section. The alignment specified in a `.align` or `.palign` directive must not be greater than the alignment implied by the section address.

S

section alignment must be a power of two.

The argument to an `align()` or `reverse()` section attribute was invalid.

section address 0xnnnn exceeds near data range .
section address must be even .
section address must be in range [0..0x7ffffe].

The argument to an `address()` section attribute was invalid.

Symbol 'name' can not be both weak and common.

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

syntax error in .startof. or .sizeof.

The assembler found either `.startof.` or `.sizeof.`, but did not find the beginning parenthesis '(' or ending parenthesis ')'. See 4.8.6. Obtaining the Size of a Specific Section and 4.8.7. Obtaining the Starting Address of a Specific Section for details on the `.startof.` and `.sizeof` operators.

T

This expression is not a valid GOTO target.

The assembler does not support expressions that include unresolved symbols as a GOTO destination.

Too few operands ('insn').

Too few operands were specified for this instruction.

Too many operands ('insn').

Too many operands were specified for this instruction.

U

unexpected end of file in irp or irpc.

The end of the file was seen before the terminating `.endr` directive.

unexpected end of file in macro definition.

The end of the file was seen before the terminating `.endm` directive.

Unknown pseudo-op: 'directive'.

The assembler does not recognize the specified directive. Check to see that you have spelled the directive correctly.

Note: The assembler expects that anything that is preceded by a dot (.) is a directive.

W

WAR hazard detected.

The assembler found a Write After Read hazard in the instruction. A WAR hazard occurs when a common W register is used for both the source and destination given that the source register uses pre/post-increment/decrement.

Word operations expect even offset.

An attempt was made to specify [Wn+offset] or [Wn-offset] where offset is even with a word instruction.

Word operations expect an even offset between -1024 and 1022.

The offset specified in [Wn+offset] or [Wn-offset] was even, but exceeded the maximum or minimum value allowed for word instructions.

7.3 Warnings

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

Symbol
.def pseudo-op used inside of .def/.endef: ignored.

The specified directive is not allowed within a `.def`/`.endef` pair. `.def`/`.endef` directives are used for specifying debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note the following:

1. you want to use the `.line` directive to specify the line number information for the symbol, and
2. you cannot nest `.def`/`.endef` directives.

.dim pseudo-op used outside of .def/.endef: ignored.

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

.endef pseudo-op used outside of .def/.endef: ignored.

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

.fill size clamped to 8.

The size argument (second argument) of the `.fill` directive specified was greater then eight. The maximum size allowed is eight.

.fillupper expects a constant positive byte value. 0xXX assumed.

The `.fillupper` directive was specified with an argument that is not a constant positive byte value. The last `.fillupper` value that was specified will be used.

.fillupper not specified in a code section. .fillupper ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.fillvalue expects a constant positive byte value. 0xXX assumed.

The `.fillvalue` directive was specified with an argument that is not a constant positive byte value. The last `.fillvalue` value that was specified will be used.

.fillvalue not specified in a code section. .fillvalue ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.ln pseudo-op inside .def/.endef: ignored.

The specified directive is not allowed within a `.def`/`.endef` pair. `.def`/`.endef` directives are used for specifying debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note the following:

1. you want to use the `.line` directive to specify the line number information for the symbol, and
2. you cannot nest `.def`/`.endef` directives.

.loc outside of .text.

The `.loc` directive must be specified in a `.text` section. The assembler has seen this directive in a non-.text section. The directive has no effect.

.loc pseudo-op inside .def/.endef: ignored.

The specified directive is not allowed within a `.def`/`.endef` pair. `.def`/`.endef` directives are used for specifying debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note the following:

1. you want to use the `.line` directive to specify the line number information for the symbol, and
2. you cannot nest `.def`/`.endef` directives.

.palign not specified in a code section. .palign ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pbyte not specified in a code section. .pbyte ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pfill not specified in a code section. .pfill ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pfill size clamped to 8.

The size argument (second argument) of the .fill directive specified was greater then eight. The maximum size allowed is eight.

.pfillvalue expects a constant positive byte value. 0xXX assumed.

The `.pfillvalue` directive was specified with an argument that is not a constant positive byte value. The last `.pfillvalue` value that was specified will be used as if this directive did not exist.

.pfillvalue not specified in a code section. .pfillvalue ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

.pword not specified in a code section. .pword ignored.

The specified directive must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

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

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

.scl pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

.tag pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

.type pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

.val pseudo-op used outside of .def/.endef ignored.

The specified directive is only allowed within a `.def`/`.endef` pair. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first specify a `.def` directive before specifying this directive.

A
Alignment too large: 2^15 assumed.

An alignment greater than 2^15 was requested. 2^15 is the largest alignment request that can be made.

B
badly formed .dim directive ignored

The arguments for the `.dim` directive were unable to be parsed. This directive is used to specify debugging information and normally is only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, the arguments for the .dim directive are constant integers separated by a comma.

D
Directive not specified in a code section. Directive ignored.

The directive on the indicated line must be specified in a code section. The assembler has seen this directive in a data section. This warning probably indicates that you forgot to change sections to a code section.

E
error setting flags for “section_name”: error_message.

If this warning is displayed, then the GNU code has changed as the if statement always evaluates false.

Expecting even address. Address will be rounded.

The absolute address specified for a CALL or GOTO instruction was odd. The address is rounded up. You will want to ensure that this is the intended result.

Expecting even offset. Offset will be rounded.

The PC-relative instruction at this line contained an odd offset. The offset is rounded up to ensure that the PC-relative instruction is working with even addresses.

I
Ignoring changed section attributes for section_name.

This section’s attributes have already been set, and the new attributes do not match those previously set.

Ignoring fill value in absolute section.

A fill argument cannot be specified for either the `.org` or `.porg` directive when the current section is absolute.

Implied attributes for section 'name' are deprecated.

Certain section names have implied attributes. In this case, a section was defined without listing its implied attributes. For clarity and future compatibility, section attributes should be listed explicitly. See 5.1. Directives that Define Sections for more information.

L

Line numbers must be positive integers.

The line number argument of the `.ln` or `.loc` directive was less than or equal to zero after specifying debugging information for a function. These directives are used to specify debugging information and normally are only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, note that function symbols can only exist on positive line numbers.

M

Macro 'name' has a previous definition.

A macro has been redefined without removing the previous definition with the `.purgem` directive.

mismatched .eb

The assembler has seen a `.eb` directive without first seeing a matching `.bb` directive. The `.bb` and `.eb` directives are the begin block and end block directives and must always be specified in pairs.

O

Overflow/underflow for .long may lose significant bits.

A constant value specified in a .long directive is too large and will lose significant bits when encoded.

Q

Quoted section flags are deprecated, use attributes instead.

Previous versions of the assembler recommended the use of single character section flags. For clarity and future compatibility, attribute names should be used instead.

R

Repeat argument < 0. .fill ignored.

The repeat argument (first argument) of the `.fill` directive specified was less than zero. The repeat argument must be an integer that is greater than or equal to zero.

Repeat argument < 0. .pfill ignored.

The repeat argument (first argument) of the `.pfill` directive specified was less than zero. The repeat argument must be an integer that is greater than or equal to zero.

S

Size argument < 0. .fill ignored.

The size argument (second argument) of the .fill directive specified was less than zero. The size argument must be an integer that is between zero and eight, inclusive. If the size argument is greater than eight, it is deemed to have a value of eight.

Size argument < 0. .pfill ignored

The size argument (second argument) of the `.pfill` directive specified was less than zero. The size argument must be an integer that is between zero and eight, inclusive. If the size argument is greater than eight, it is deemed to have a value of eight.

'symbol_name' symbol without preceding function.

A `.bf` directive was seen without the preceding debugging information for the function symbol. This directive is used to specify debugging information and normally is only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must first .def the function symbol and give it a .type of function (C_FCN = 101).

T

tag not found for .tag symbol_name.

This warning should not be seen unless the assembler was unable to create the given symbol name. Check your code for errors. If you still receive this warning, contact technical support.

U
unexpected storage class sclass.

The assembler is processing the `.endef` directive and has either seen a storage class that it does not recognize or has not seen a storage class. This directive is used to specify debugging information and normally is only generated by the compiler. If you are attempting to specify debugging information for your assembly language program, you must specify a storage class using the `.scl` directive, and that storage class cannot be one of the following:

1. Undefined static (C_USTATIC = 14)
2. External definition (C_EXTDEF = 5)
3. Undefined label (C_ULABEL = 7)
4. Dummy entry (end of block) (C_LASTENT = 20)
5. Line # reformatted as symbol table entry (C_LINE = 104)
6. Duplicate tag (C_ALIAS = 105)
7. External symbol in dmert public library (C_HIDDEN = 106)
8. Weak symbol - GNU extension to COFF (C_WEAKEXT = 127)

unknown section attribute 'flag'.

The .section directive does not recognize the specified section flag. Please see 5.1. Directives that Define Sections, for the supported section flags.

unsupported section attribute 'i'.

The .section directive does not support the "i" section flag for COFF. Please see 5.1. Directives that Define Sections, for the supported section flags.

unsupported section attribute 'l'.

The .section directive does not support the "l" section flag for COFF. Please see 5.1. Directives that Define Sections, for the supported section flags.

unsupported section attribute 'o'.

The .section directive does not support the "o" section flag for COFF. Please see 5.1. Directives that Define Sections, for the supported section flags.

V
Value get truncated to use.

The fill value specified for either the `.skip`, `.pskip`, `.space`, `.pspace`, `.org` or `.porg` directive was larger than a single byte. The value has been truncated to a byte.

7.4 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.

8. Linker Overview

MPLAB XC16 Object Linker produces binary code from relocatable object code, and any archive/library files, for the dsPIC® DSC and PIC24 MCU families of devices. The 16-bit linker is an application that provides a platform for developing executable code. The linker is a part of the GNU linker from the Free Software Foundation.

8.1 Feature Set

Notable features of the linker include:

- Automatic or user-defined stack allocation
- Supports 16-bit Program Space Visibility (PSV) window
- Available for Windows, Linux and Mac OS
- Command Line Interface
- Linker scripts for all 16-bit devices
- Available for MPLAB® X IDE

8.2 Linker Usage

The MPLAB XC16 Object Linker translates object files from the MPLAB XC16 assembler, and archive/library files from the MPLAB XC16 archiver/librarian, into an executable file. See the "*MPLAB XC16 C Compiler User's Guide*" (DS50002071) for an overview of the tools process flow.

In most instances it will not be necessary to invoke the linker directly, as the compiler driver, `xc16-gcc`, will automatically execute the linker with all necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the compiler and linking in general. The compiler often makes assumptions about the way in which the program will be linked. If the linker sections are not linked correctly, code failure may result.

8.3 Input/Output Files

Linker input and output files are listed below.

Table 8-1. Linker Files

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

The MPLAB XC16 linker does not generate absolute listing files. The 16-bit linker is capable of creating a map file and a binary file (that may or may not contain debugging information).

8.3.1 Object Files

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

```
xc16-ld -omf=elf ...
```

Alternatively, the environment variable XC16_OMF may be used to specify object file format for the 16-bit language tools.

8.3.2 Library Files

A collection of object files grouped together for convenience.

8.3.3 Linker Script File

Linker scripts, or command files:

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

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

Example 8-1. Linker Script

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

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

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

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

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

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

8.3.4 Linker Output File

By default, the name of the linker output binary file is `a.out`. You can override the default name by specifying the `-o` option on the command line. The format of the binary file is an executable ELF file by default. To specify a ELF or COFF executable file, use the `-omf` option as shown in Section 7.3.1 “Object Files.”

8.3.5 Map File

The linker has the capability to produce map files. For details on how to generate a map file and the components of that file, see Chapter 12. “Linker Map File.”

9. Linker Command Line Options

MPLAB XC16 Object Linker may be used on the command line interface as well as with an IDE.

9.1 Syntax

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

```
xc16-ld [options] file...
```

Note: Command line options are case sensitive.

For example, `xc16-ld` links object files and archives to produce a binary file. To link a file `hello.o`:

```
xc16-ld -o output hello.o -lpic30
```

This tells `xc16-ld` to produce a file called `output` as the result of linking the file `hello.o` with the archive `libpic30.a`.

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

```
xc16-ld -o output hello.o --start-group -lpic30 -lm -lc --end-group
```

The command line options to `xc16-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 (i.e., those farther 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` (lowercase 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 with names that 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 with names that 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 equal 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.

9.2 Options that Control Output File Creation

Output file creation options are described below.

9.2.1 `--add-flags-code=,--add-flags-data=,--add-flags-const=`

Add section attributes to all code, data, or PSV (`.const`) type sections.

Each option can be repeated or have a comma-separated list of additions. The flag names are the same as you would find in the assembler documentation, with one exception: '`name()`' is used to prepend a name to the currently-provided section name.

```
--add-flags-code=attr1,attr2,attr3
--add-flags-code=attr1 --add-flags-code=attr2 --add-flags-code=attr3
--add-flags-code=name(fred)
```

`=name(x)` will prepend the current section name with `x`. This allows 'sequentially' mapped sections to be mapped without having to modify the linker script, i.e., to add '`boot`' to all text type sections:

```
--add-flags-code=name(remapped) --add-flags-code=boot
```

Any incompatible section will be flagged.

9.2.2 --application-id=*name*

Used for co-residency. Specify an application ID for the application code being compiled. This will define extra symbols using the application ID symbol name. For example:

```
xc16-gcc -DVERSION=1 foo.c -o foo.exe -Wl,--application-id=foo
```

where symbol `_version` would be mapped to `_foo_version`.

9.2.3 --architecture arch (-A arch)

Set architecture.

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

9.2.4 -(archives -), --start-group archives, --end-group

Start and end a group.

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

9.2.5 --coresident

Co-residency Linking.

Inform the linker that a coresident link is being performed and to omit the Reset vector from a link. Used with `--no-isr.`

Example:

```
$ xc16-gcc example.c -mcpu=30f6014 -T p30F6014.gld -o
example.exe -Wl,--no-isr,--coresident
```

9.2.6 -d, -dc, -dp

Force common symbols to be defined.

Assign space to common symbols even when a relocatable output file is specified (i.e., with `-r`).

9.2.7 --defsym *sym=expr*

Define a symbol.

Create a global symbol in the output file that contains 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*.

9.2.8 --discard-all (-x)

Discard all local symbols.

9.2.9 --discard-locals (-X)

Discard temporary local symbols.

9.2.10 --fill=*option*

Fill unused program memory. The format is:

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

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

Single value

```
xc16-ld --fill=0x12345678@unused
```

Range of values

```
xc16-ld --fill=1,2,3,4,097@0x9d000650:0x9d000750
```

Incrementing value

```
xc16-ld --fill=7+=911@unused
```

By default, the linker will fill using data that is instruction-word length. For 16-bit devices, the default fill width is 24 bits. However, you may specify the value width using `[wn:]`, where w is the fill value's width and n belongs to [1, 3].

Multiple fill options may be specified on the command line; the linker will always process fill options at specific locations first.

9.2.11 --fill-upper *value*

Set fill value for upper byte of data.

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

9.2.12 --force-exe-suffix

Force generation of file with `.exe` suffix.

9.2.13 --force-link

Force linking of objects that may not be compatible.

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

9.2.14 --no-force-link

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

9.2.15 --gc-sections

Remove unused (dead) functions from code at link time.

Support is for ELF projects only. In order to make the best use of this feature, add the `-ffunction-sections` option to the compiler command line.

9.2.16 --isr

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

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

9.2.17 --no-isr

Do not create an interrupt function for vectors unused by the application.

Do not create a default interrupt function if an application does not provide one.

The unused vector slots will remain unfilled and can be defined in a future link (as in co-resident applications).

9.2.18 --ivt

The linker is instructed to generate an IVT or AIVT, unless one is explicitly created in the linker script or by other means.

9.2.19 --no-ivt

The linker is instructed *not* to generate an IVT or AIVT, unless one is explicitly created in the linker script or by other means.

9.2.20 -legacy-libc

Use legacy include files and libraries (those distributed with v3.24 and before).

The content of include file and libraries changed in v3.25 to be compatible with the HI-TECH C compiler.

9.2.21 --library *libname* (-l *libname*)

Search for library *libname*.

Add archive file *libname* to the list of files to link. This option may be used any number of times. `xc16-ld` will search its path-list for occurrences of `lib`*libname*`.a` for every *libname* specified. The linker will search an archive only once, at the location where it is specified on the command line. If the archive defines a symbol that was undefined in some object that appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again. See the `-(` option for a way to force the linker to search archives multiple times. You may list the same archive multiple times on the command line.

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

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

9.2.22 --library-path <dir> (-L <dir>)

Add <dir> to library search path.

Add path <dir> to the list of paths that `xc16-ld` will search for archive libraries and `xc16-ld` control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. All `-L` options apply to all `-l` options, regardless of the order in which the options appear. The library paths can also be specified in a link script with the `SEARCH_DIR` command. Directories specified this way are searched at the point in which the linker script appears in the command line.

9.2.23 --msecondary-id, --msecondary-id-location

--msecondary-id= is used to assign a unique ID to a secondary executable. This can be used when there are multiple secondary applications in a single project and the main core may dynamically select which image is programmed. Giving these applications a unique ID will allow the IDE to select the correct image during debugging. By default the linker will allocate this location at the end of memory; using --msecondary-id-location= allows selection of this location.

9.2.24 -mreserve

The link shell will process this option by creating a temporary linker script that will define a section to reserve a certain range of memory:

[start_address:end_address].

Syntax:

```
-mreserve=memory_region@start_address:end_address
```

The section created will have the name pattern:

```
reserve_ memory_region_ start_address
```

The address of the section will be: start_address

The length of the section will be: end_address - start_address + 1

Example:

```
-mreserve=data@0x1000:0x1050
```

A temporary linker script with the following content will be created:

```
SECTIONS {
reserve_data_0x1000 0x1000: {
SHORT(0x0);
. = 0x51;
}
}
```

Multiple -mreserve options can be passed to the linker to reserve multiple ranges.

9.2.25 --no-keep-memory

Use less memory and more disk I/O.

xc16-ld normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. This option tells xc16-ld to instead optimize for memory usage by rereading the symbol tables, as necessary. This may be required if xc16-ld runs out of memory space while linking a large executable.

9.2.26 --noinhibit-exec

Create an output file even if errors occur.

Retain the executable output file whenever it is still usable. Normally, when the linker encounters an error during the link process, it will exit without writing an output file.

9.2.27 -omf=*format*

xc16-ld produces ELF format output binary files by default. Use this option to specify ELF or COFF format explicitly. Alternatively, the environment variable XC16_OMF may be used to specify object file format for the 16-bit language tools.

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

9.2.28 --output file (-o file)

Set output file name.

Use file as the name for the program produced by `xc16-ld`; if this option is not specified, the name `a.out` is used by default.

9.2.29 --pad-flash=*size*

Ensure that the linked output section is padded to a *size* byte boundary.

Used with co-resident applications.

9.2.30 --preserve=*executable*

Use a previously-compiled executable to identify where to allocate preserved variables.

9.2.31 --preserve-all

Preserve all variables unless explicitly marked with update.

9.2.32 -p,--processor *PROC*

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

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

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

Generate relocatable output.

That is, generate an output file that can, in turn, serve as input to `xc16-ld`. This is often called partial linking. If this option is not specified, an absolute file is produced.

9.2.34 --reserve-const=*size*

Reserve the specified amount of const data (size). If no value is specified, the maximum is reserved.

Used with co-resident applications.

9.2.35 --retain-symbols-file *file*

Keep only symbols listed in *file*.

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

9.2.36 --script file (-T *file*)

Read linker script.

Read link commands from the file *file*. These commands replace the default link script of `xc16-ld` (rather than adding to it), so file must specify everything necessary to describe the target format. If file does not exist, `xc16-ld` looks for it in the directories specified by any preceding `-L` options. Multiple `-T` options accumulate.

9.2.37 --select-objects

Select library objects based on options (this is the default).

Some compiler options, such as `-mlarge-arrays`, must be set consistently across all objects in an application. In order to maintain full compatibility, pre-compiled libraries must contain multiple versions of each object. Library objects are selected based on a signature which is created by the compiler and reflects the options used to create the object. Objects from older libraries that lack a signature are considered to be compatible if the restrictive compiler options have not been set.

9.2.38 --no-select-objects

Don't select library objects based on options.

This option causes the linker to load the first instance of a library object, regardless of the options signature. This option can be used to force library compatibility with restrictive compiler options, even if the library lacks a signature.

9.2.39 --smart-io

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

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

9.2.40 --no-smart-io

Don't merge I/O library functions.

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

9.2.41 --strip-all (-s)

Strip all symbols.

Omit all symbol information from the output file.

9.2.42 --strip-debug (-S)

Strip debugging symbols.

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

9.2.43 -Tbss *address*

Set address of `.bss` section.

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

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

9.2.44 -Tdata *address*

Set address of `.data` section.

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

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

9.2.45 -Ttext *address*

Set address of `.text` section.

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

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

9.2.46 --undefined symbol (-u *symbol*)

Start with undefined reference to *`symbol`*.

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

9.2.47 --no-undefined

Allow no undefined symbols.

9.2.48 --wrap *symbol*

Use wrapper functions for *symbol*.

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

Here is an example:

```
#include <stdio.h>
#ifdef WRAP
#define WRAPIT2(X) _wrap__ ## X
#define WRAPIT(X) WRAPIT2(X)
#define REAL2(X) _real__ ## X
#define REAL(X) REAL2(X)
#else
#define WRAPIT(X) X
#define REAL(X) X
#endif
int REAL(foo)(int x) {
/* this is the real function */
/* we if we want to wrap it, then this is called _real__foo (or __real_foo from the linkers
perspective
*/
return x +1;
}
#ifdef WRAP
/* wrap it and call the real one */
int WRAPIT(foo)(int x) {
return REAL(foo)(x)+1;
}
#endif
main() {
fprintf(stderr,"foo returns %d\n", foo(1));
}
```

The following will wrap foo and make it call __wrapfoo; __real_foo will call the real foo function. The output will be 3.

```
xc16-gcc wrap.c -save-temps -DWRAP -Wl,--wrap,_foo
```

The following will not wrap foo, and make the 'real' foo be simply foo. The output will be 2.

```
xc16-gcc wrap.c -save-temps
```

The --wrap option can be used to intercept a precompiled library call. Be aware that this option requires an assembly level symbol name. For example, in order to wrap the C symbol foo, you must specify --wrap _foo, since all C symbols will be given a leading underscore by the C compiler.

9.3 Options that Control Run-time Initialization

Run-time initialization options are described below.

9.3.1 --data-init

Support initialized data (this is the default).

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

9.3.2 --no-data-init

Don’t support initialized data.

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

9.3.3 --handles

Support far code pointers (this is the default).

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

9.3.4 --no-handles

Don’t support far code pointers.

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

9.3.5 --heap size

Set heap to size bytes.

Allocate a run-time heap of size bytes for use by C programs. The heap is allocated from unused data memory. If sufficient memory is unavailable, an error is reported.

9.3.6 --local-stack

Prevent allocating the stack in extended data space memory (this is the default).

9.3.7 --no-local-stack

Allow allocating the stack in extended data space memory.

9.3.8 --pack-data

Pack initial data values (this is the default).

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

9.3.9 --no-pack-data

Don’t pack initial data values.

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

9.3.10 --stack size

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

By default, the linker allocates all unused data memory for the run-time stack. Alternatively, the programmer may allocate the stack by defining a section with the `stack` attribute. Use this option to ensure that at least a minimum-sized stack is available. The actual stack size is reported in the link map output file. If the minimum size is not available, an error is reported. The default minimum stack size does not include a stack guardband, as described in the next section.

9.3.11 --stackguard *size*

Set stack guardband to size bytes (default=16).

By default a portion of the physical stack is reserved for a guardband.

The stack guardband ensures that enough stack space is available to process a stack overflow exception. The default value (16 bytes) was chosen to handle the worst-case scenario, and guarantees that an exception handler can be invoked. This option can be used to reserve additional stack space for exception processing, or to reduce the guardband size, freeing up additional memory for the stack.

9.4 Options that Control Informational Output

Information output options are described below.

9.4.1 --check-sections

Check section addresses for overlaps (this is the default).

9.4.2 --no-check-sections

Do not check section addresses for overlaps.

9.4.3 --help

Print option help.

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

9.4.4 --memory-usage

Specify FLASH and Data memory usage. Useful for co-resident applications.

Enables the writing of two data tables, one for FLASH and one for data memory. Each table is `NULL` terminated. Each ROW contains a pair of values in FLASH, the first is the start address of the consumed memory and the second is the last address.

The FLASH table can be accessed via the global symbol `__ROM_USAGE` and the RAM table can be accessed with the global symbol `__RAM_USAGE`. The symbols can be accessed using `tblrd` instructions or any other access method that can read the upper byte of FLASH.

9.4.5 --no-psrd-psrd-check

This is a linker option that can be used to disable the automatic check for PSRD PSRD violations (back-to-back data flash reads). In general, it is not recommended that this option be used.

9.4.6 --no-warn-mismatch

Do not warn about mismatched input files.

Normally `xc16-ld` will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells `xc16-ld` that it should silently permit such possible errors. This option should only be used with care in cases when you have taken some special action that ensures that the linker errors are inappropriate.

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

9.4.7 --report-mem

Print a memory usage report.

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

9.4.8 --trace (-t)

Trace file.

Print the names of the input files as `xc16-ld` processes them.

9.4.9 --trace-symbol *symbol* (-y *symbol*)

Trace mentions of *symbol*.

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

9.4.10 -v

Print version and other information.

9.4.11 --verbose

Output lots of information during link.

Display the version number for `xc16-ld`. Display the input files that can and cannot be opened. Display the linker script if using a default built-in script.

9.4.12 --version (-v)

Print version information.

9.4.13 --warn-common

Warn about duplicate common symbols.

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

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

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

```
int i = 1;
```

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

```
extern 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.

```
int i;
```

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
```

9.4.14 --warn-once

Warn only once per undefined symbol.

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

9.4.15 --warn-section-align

Warn if start of section changes due to alignment.

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

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

9.5 Options that Modify the Link Map Output

Link map output modifying options are described below.

9.5.1 --cref

Output cross-reference table.

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

9.5.2 --print-map (-M)

Print map file on standard output.

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

Where object files and symbols are mapped into memory.

How common symbols are allocated.

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

9.5.3 -Map *file*

Write a map file.

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

9.6 Options that Specify CodeGuard™ Security Features

Three linker options are related to CodeGuard Security:

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

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

9.6.1 CodeGuard Security Segment Options

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

For MPLAB X IDE, these options will appear in the Project Properties window under xc16-ld options. They will be passed to the linker via command line.

Table 9-1. CodeGuard™ Security Segment Options

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

.....continued			
Option	**Segment(s) Supported**		
	boot	**secure**	**general**
code_protect_high			X
** default setting			

CodeGuard Security segment options

```
--boot small_flash_std
--boot=small_ram:medium_flash_std:eeprom
--secure no_ram:small_flash_std
--secure=medium_ram:large_flash_high
--general write_protect
--general=no_write_protect:code_protect_high
```

9.6.2 User-Defined Segment Options

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

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

Table 9-2. User-Defined Segment Options

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

User-Defined segment options

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

9.7 Options that Control the Preprocessor

Linker scripts are passed to the C preprocessor before actual linking begins. This provides an opportunity to substitute macro definitions and to include conditional blocks of code. The C preprocessor is well-known by programmers and documentation is widely available.

Linker preprocessor options are listed in the sections below.

9.7.1 -D<macro>[=*value*]

Define a macro (with optional value) to the preprocessor.

Macros can be used to substitute literal values into a script, such as for the origin or length of memory regions. They can also be used to select conditional blocks of code using directives such as `#ifdef`, `#endif`.

9.7.2 --no-cpp

Do not preprocess linker scripts.

Linker script preprocessing is enabled by default. This option can be used to disable preprocessing.

Care should be used in selecting this option. If a linker script requires preprocessing (such as for conditional blocks of text), using this option will cause a processing error.

9.7.3 `--save-gld`

Save preprocessed linker scripts.

By default the result of preprocessing is a temporary file. This option can be used to save the preprocessed linker script. A filename is automatically generated based on the linker script filename.

Notes:

10. Linker Scripts

Linker scripts are used to control MPLAB XC16 Object Linker functions. You can customize your linker script for specialized control of the linker.

10.1 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)
- assignment of absolute SFR addresses for the target device

10.1.1 Contents

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

10.1.2 File Names and Locations

The 16-bit Language Tools include a set of standard linker scripts: device-specific linker scripts (e.g., `p30f3014.gld`) and one generic linker script (`p30sim.gld`).

Standard linker script files are provided for each device and are located under:

```
Install_Dir/support/DeviceFamily/gld
```

where *`Install_Dir`* in the installation directory for the MPLAB XC16 C compiler and *`DeviceFamily`* is be the name of the device family (e.g., dsPIC33EP) or generic.

10.2 Command Line Information

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

```
xc16-ld -o output.cof output.o --script
  ..\support\dsPIC30F\gld\p30f3014.gld
```

If the linker is invoked through `xc16-gcc`, add the `-Wl,` prefix to allow the option to be passed to the linker:

```
xc16-gcc -o output.cof output.s -Wl,--script,
..\support\dsPIC30F\gld\p30f3014.gld
```

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

```
xc16-ld --verbose
```

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

Linker scripts are located by using the library search path which, by default, includes the standard directories provided with the install.

10.3 Contents of a Linker Script

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

10.3.1 Processor and Startup Modules

The first several lines of a linker script define the processor and startup modules:

```
/*
** Linker Script for 30f3014
*/
OUTPUT_ARCH("30F3014")
CRT0_STARTUP(crt0_standard.o)
CRT1_STARTUP(crt1_standard.o)
OPTIONAL(-lp30F3014)
```

The `OUTPUT_ARCH` command specifies the target processor. The `CRTn_STARTUP` commands specify two C run-time startup modules to be loaded from archives. The linker will select one of these based on whether data initialization has been enabled. The `OPTIONAL` command specifies a device-specific library that should be opened if available. If the library file cannot be found, the link will continue without error unless there are unresolved references in the application.

10.3.2 Memory Region Information

The next section of a linker script defines the various memory regions for the target device using the `MEMORY` command.

For the dsPIC30F3014, several memory regions are defined:

```
/*
** Memory Regions
*/
MEMORY
{
 data          : ORIGIN = 0x800,    LENGTH = 2048
 program       : ORIGIN = 0x100,    LENGTH = ((8K * 2) - 0x100)
 reset         : ORIGIN = 0,        LENGTH = (4)
 ivt           : ORIGIN = 0x04,     LENGTH = (62 * 2)
 aivt          : ORIGIN = 0x84,     LENGTH = (62 * 2)
 __FOSC        : ORIGIN = 0xF80000, LENGTH = (2)
 __FWDT        : ORIGIN = 0xF80002, LENGTH = (2)
 __FBORPOR     : ORIGIN = 0xF80004, LENGTH = (2)
 __CONFIG4     : ORIGIN = 0xF80006, LENGTH = (2)
 __CONFIG5     : ORIGIN = 0xF80008, LENGTH = (2)
 __FGS         : ORIGIN = 0xF8000A, LENGTH = (2)
 __FUID0       : ORIGIN = 0x8005C0, LENGTH = (2)
 __FUID1       : ORIGIN = 0x8005C2, LENGTH = (2)
 __FUID2       : ORIGIN = 0x8005C4, LENGTH = (2)
 __FUID3       : ORIGIN = 0x8005C6, LENGTH = (2)
 eedata        : ORIGIN = 0x7FFC00, LENGTH = (1024)
}
```

Each memory region is range-checked as sections are added during the link process. If any region overflows, a link error is reported.

`MEMORY` regions are shown below.

10.3.2.1 Data Region

```
data          : ORIGIN = 0x800,     LENGTH = 2048
```

The data region corresponds to the RAM memory of the dsPIC30F3014 device, and is used for both initialized and uninitialized variables. The starting address of region `data` is `0x800`. This is the first usable location in RAM, after the space reserved for memory-mapped SFRs.

10.3.2.2 Program Region

```
program      : ORIGIN = 0x100, LENGTH = ((8K * 2) - 0x100)
```

The program region corresponds to the Flash memory of the dsPIC30F3014 device that is available for user code, library code and constants. The starting address of region program is `0x100`. This is the first location in Flash that is available for general use. Addresses below `0x100` are reserved for the Reset instruction and the two vector tables.

The length specification of the program region deserves particular emphasis. The (8K * 2) portion indicates that the dsPIC30F3014 has 8K instruction words of Flash memory, and that each instruction word is 2 address units wide. The `- 0x100` portion reflects the fact that some of the Flash is reserved for the Reset instruction and vector tables.

Note: Instruction words in the 16-bit devices are 24 bits, or 3 bytes, wide. However, the PC increments by 2 for each instruction word for compatibility with data memory. Address and lengths in program memory are expressed in PC units.

10.3.2.3 Reset, Ivt and Aivt Regions

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

The Reset region corresponds to the 16-bit Reset instruction at address `0` in program memory. The Reset region is 4 address units, or 2 instruction words, long. This region always contains a `GOTO` instruction that is executed upon device reset. The `GOTO` instruction is encoded by data commands in the section map (see 10.3.4.1. Output Section .reset).

```
ivt          : ORIGIN = 0x04,      LENGTH = (62 * 2)
aivt         : ORIGIN = 0x84,      LENGTH = (62 * 2)
```

The `ivt` and `aivt` regions correspond to the interrupt vector table and alternate interrupt vector table, respectively. Each interrupt vector table contains 62 entries, each is 2 address units in length. Each entry represents a word of program memory, which contains a 24-bit address. The linker initializes the vector tables with appropriate data, according to standard naming conventions.

Regions `reset`, `ivt` and `aivt` comprise the low address portion of Flash memory that is not available for user programs.

10.3.2.4 Fuse Configuration Regions

```
__FOSC       : ORIGIN = 0xF80000, LENGTH = (2)
__FWDT       : ORIGIN = 0xF80002, LENGTH = (2)
__FBORPOR    : ORIGIN = 0xF80004, LENGTH = (2)
__CONFIG4    : ORIGIN = 0xF80006, LENGTH = (2)
__CONFIG5    : ORIGIN = 0xF80008, LENGTH = (2)
__FGS        : ORIGIN = 0xF8000A, LENGTH = (2)
```

These regions correspond to the dsPIC30F3014 configuration registers.

Each fuse configuration region is exactly one instruction word long. If sections are defined in the application source code with the standard naming convention, the section contents will be written into the appropriate configuration register(s). Otherwise, the registers are left uninitialized. If more than one value is defined for any configuration region, a link error will be reported.

10.3.2.5 Unit ID Regions

```
__FUID0      : ORIGIN = 0x8005C0, LENGTH = (2)
__FUID1      : ORIGIN = 0x8005C2, LENGTH = (2)
__FUID2      : ORIGIN = 0x8005C4, LENGTH = (2)
__FUID3      : ORIGIN = 0x8005C6, LENGTH = (2)
```

The unit ID regions correspond to locations in program memory that may be programmed with application-specific information.

10.3.2.6 EEDATA Memory Region

```
eedata       : ORIGIN = 0x7FFC00,  LENGTH = (1024)
```

The eedata region corresponds to non-volatile data flash memory located in high memory. Although located in program memory space, the data flash is organized like data memory. The total length is 1024 bytes.

10.3.3 Base Memory Addresses

This portion of the linker script defines the base addresses of several output sections in the application. Each base address is defined as a symbol with the following syntax:

```
name = value;
```

The symbols are used to specify load addresses in the section map. For the dsPIC30F3014, several base memory addresses are defined. Not all of these symbols are referenced in the section map; some are included for informational purposes.

```
/*
** Base Memory Addresses - Program Memory
*/
__RESET_BASE  = 0;        /* Reset Instruction               */
__IVT_BASE    = 0x04;     /* Interrupt Vector Table          */
__AIVT_BASE   = 0x84;     /* Alternate Interrupt Vector Table */
__CODE_BASE   = 0x100;    /* Handles, User Code, Library Code */
/*
** Base Memory Addresses - Data Memory
*/
__SFR_BASE    = 0;        /* Memory-mapped SFRs                */
__DATA_BASE   = 0x800;    /* X and General Purpose Data Memory  */
__YDATA_BASE  = 0x0C00;   /* Y Data Memory for DSP Instructions */
```

10.3.4 Input/Output Section Map

The section map is the heart of the linker script. It defines how input sections are mapped to output sections. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. Generally, several input sections may be combined into a single output section.

For example, suppose that an application is comprised of five different functions, and each function is defined in a separate source file. Together, these source files will produce five input sections. The linker will combine these input sections into a single output section. Only the output section has an absolute address.

If any input or output sections are empty, there is no penalty or storage cost for the linked application. Most applications will use only a few of the many sections that appear in the section map.

10.3.4.1 Output Section .reset

Section .reset contains a GOTO instruction, created at link time, from output section data commands:

```
/*
** Reset Instruction
*/
.reset __RESET_BASE :
{
 SHORT(ABSOLUTE(__reset));
 SHORT(0x04);
 SHORT((ABSOLUTE(__reset) >> 16) & 0x7F);
 SHORT(0);
} >reset
```

Each SHORT() data command causes a 2 byte value to be included. There are two expressions which include the symbol __reset, which by convention is the first function invoked after a device reset. Each expression calculates a portion of the address of the Reset function. These declarations encode a 24-bit GOTO instruction, which is two instruction words long.

The ABSOLUTE() function specifies the final value of a program symbol after linking. If this function were omitted, a relative (before-linking) value of the program symbol would be used.

The >reset portion of this definition indicates that this section should be allocated in the Reset memory region.

10.3.4.2 Output Section .text

Section `.text` collects executable code from all of the application's input files.

```
/*
** User Code and Library Code
*/
.text :
{
 *(.init);
 *(.user_init);
 keep(*(.handle));
 keep(*(.isr));
 *(.libc) *(.libm) *(.libdsp); /* keep together in this order */
 *(.lib*);
} >program
```

Several different input sections are collected into one output section. This was done to ensure the order in which the input sections are loaded.

Table 10-1. Section Types and Names

Section Type	Section Name	Description
input	`.init`	Contains the startup code that is executed immediately after device reset. It is positioned first so that its address may be readily available.
input	`.user_init`	Contains a call table for user initialization functions.
input	`.handle`	Used for function pointers and is loaded first at low addresses. `keep` is required to prevent `-gcc-sections` from deleting this code.
input	`.isr`	Used for interrupt service functions. Again, `keep` is used to preserve the code.

library	`.libc`
	`.libm`
	`.libdsp`
		These sections must be grouped together to ensure locality of reference.

library	`.lib*`	Collects other libraries, such as the peripheral libraries (which are allocated in section `.libperi`).

The input section `.text` is not explicitly mapped so that the linker may distribute code around PSV sections in order to more successfully satisfy PSV address requirements.

10.3.4.3 User-Defined Section in Program Memory

A stub is included for user-defined output sections in program memory. This stub may be edited as needed to support the application requirements. Once a standard linker script has been modified, it is called a "custom linker script." In practice, it is often simpler to use section attributes in source code to locate user-defined sections in program memory. See 12. Linker Examples for more information.

```
/*
** User-Defined Section in Program Memory
**
** note: can specify an address using
**       the following syntax:
**
**       usercode 0x1234 :
**         {
**           *(usercode);
**         } >program
*/
usercode :
{
 *(usercode);
} >program
```

An exact, absolute starting address can be specified, if necessary. If the address is greater than the current location counter, the intervening memory space will be skipped and filled with zeros. If the address is less than the current

location counter, a section overlap will occur. Whenever two output sections occupy the same address range, a link error will be reported. Overlapping sections in program memory can not be supported.

Note: Each memory region has its own location counter.

10.3.4.4 User-Defined Constants in Program Memory

A comment block is included that describes how to define sections that will be accessed via the PSV window or the EDS window. Such sections are defined with the `psv` attribute. The syntax used to represent a PSV section address is different from other type sections. In particular, the Load Memory Address (LMA) should be defined, not the Virtual Memory Address (VMA). The LMA is unique and describes where the section is located in program memory. The VMA describes a location in the data window that may be shared by multiple pages of program memory, and is therefore not unique.

```
/*
** User-Defined Constants in Program Memory
**
** For PSV-type sections, the Load Memory Address (LMA)
** should be specified as follows:
**
**       userconst : AT(0x1234)
**         {
**           *(userconst);
**         } >program
**
** Note that mapping PSV sections in linker scripts
** is not generally recommended.
**
** Because of page alignment restrictions, memory is
** often used more efficiently when PSV sections
** do not appear in the linker script.
**
** For more information on memory allocation,
** please refer to chapter 10, "Linker Processing"
** in the Assembler, Linker manual (DS50001317).
*/
```

As noted, defining PSV-type sections in the linker script is not generally recommended. This is because sections that appear in the linker script are allocated sequentially, and PSV sections have significant page alignment restrictions. For more information on memory allocation and PSV sections, see Chapter 10. “Linker Processing.”

10.3.4.5 Output Sections in Configuration Memory

Several sections are defined that match the Fuse Configuration memory regions:

```
/*
** Configuration Fuses
*/
__FOSC :
{ *(__FOSC.sec)     } >__FOSC
__FWDT :
{ *(__FWDT.sec)     } >__FWDT
__FBORPOR :
{ *(__FBORPOR.sec) } >__FBORPOR
__CONFIG4 :
{ *(__CONFIG4.sec) } >__CONFIG4
__CONFIG5 :
{ *(__CONFIG5.sec) } >__CONFIG5
__FGS :
{ *(__FGS.sec)      } >__FGS
__FICD :
{ *(__FICD.sec)     } >__FICD
__FUID0 :
{ *(__FUID0.sec)    } >__FUID0
__FUID1 :
{ *(__FUID1.sec)    } >__FUID1
__FUID2 :
{ *(__FUID2.sec)    } >__FUID2
__FUID3 :
{ *(__FUID3.sec)    } >__FUID3
```

The Configuration Fuse sections are supported by macros defined in the 16-bit device-specific include files in `support/inc` and the C header files in `support/h`.

For example, to disable the Watchdog Timer in assembly language:

```
.include "p30f6014.inc"
config __FWDT, WDT_OFF
```

The equivalent operation in C would be:

```
#include "p30f6014.h"
_FWDT(WDT_OFF);
```

Configuration macros have the effect of changing the current section. In C, the macro should be used outside of any function. In assembly language, the macro should be followed by a `.section` directive.

10.3.4.6 User-Defined Section in Data Flash Memory

A stub is included for user-defined output sections in EEDATA memory. This stub may be edited as needed to support the application requirements. Once a standard linker script has been modified, it is called a “custom linker script.” In practice, it is often simpler to use section attributes in source code to locate user-defined sections in data flash memory. See Chapter 11. “Linker Examples.” for more information.

```
/*
** User-Defined Section in Data Flash Memory
**
** note: can specify an address using
**       the following syntax:
**
**       eedata 0x7FF100 :
**         {
**           *(eedata);
**         } >eedata
*/
eedata :
{
 *(eedata);
} >eedata
```

An exact, absolute starting address can be specified, if necessary. If the address is greater than the current location counter, the intervening memory will be skipped and filled with zeros. If the address is less than the current location counter, a section overlap will occur. Whenever two output sections occupy the same address range, a link error will reported. Overlapping sections in EEDATA memory can not be supported.

Note: Each memory region has its own location counter.

10.3.4.7 In-Circuit Debugger Memory

An in-circuit debugger/emulator requires a portion of data memory for its variables and stack. Since the debugger is linked separately and in advance of user applications, the block of memory must be located at a fixed address and dedicated for use by the debugger.

```
/*
** ICD Debug Exec
**
** This section provides optional storage for
** the in-circuit debugger. Define a global symbol
** named __ICD2RAM to enable the debugger. This section
** must be loaded at data address 0x800.
*/
.icd __DATA_BASE (NOLOAD):
{
 . += (DEFINED (__ICD2RAM) ? 0x50 : 0 );
} > data
```

Section `.icd` is designed to optionally reserve memory for the in-circuit debugger/emulator. If global symbol `__ICD2RAM` is defined at link time, `0x50` bytes of memory at address `0x800` will be reserved. The (`NOLOAD`) attribute indicates that no initial values need to be loaded for this section. The name for this symbol was created when there was only one in-circuit debugger, the MPLAB ICD 2.

10.3.4.8 User-Defined Section in Data Memory

A stub is included for user-defined output sections in data memory. This stub may be edited as needed to support the application requirements. Once a standard linker script has been modified, it is called a “custom linker script.” In practice, it is often simpler to use section attributes in source code to locate user-defined sections in data memory. See Chapter 11. “Linker Examples.” for more information.

```
/*
** User-Defined Section in Data Memory
**
** note: can specify an address using
**       the following syntax:
**
**       userdata 0x1234 :
**         {
**           *(userdata);
**         } >data
*/
userdata :
{
 *(userdata);
} >data
```

An exact, absolute starting address can be specified, if necessary. If the address is greater than the current location counter, the intervening memory space will be skipped and filled with zeros. If the address is less than the current location counter, a section overlap will occur. Whenever two output sections occupy the same address range, a link error will be reported. Overlapping sections in data memory cannot be supported.

10.3.5 Interrupt Vector Tables

The primary and alternate interrupt vector tables are defined in a second section map, near the end of the standard linker script:

```
/*
** Section Map for Interrupt Vector Tables
*/
SECTIONS
{

/*
** Primary Interrupt Vector Table
*/
.ivt __IVT_BASE :
  {
    LONG(DEFINED(__ReservedTrap0)  ? ABSOLUTE(__ReservedTrap0)  :
         ABSOLUTE(__DefaultInterrupt));
    LONG(DEFINED(__OscillatorFail) ? ABSOLUTE(__OscillatorFail) :
         ABSOLUTE(__DefaultInterrupt));
    LONG(DEFINED(__AddressError)   ? ABSOLUTE(__AddressError)   :
         ABSOLUTE(__DefaultInterrupt));
    :
    :
    LONG(DEFINED(__Interrupt53)    ? ABSOLUTE(__Interrupt53)    :
         ABSOLUTE(__DefaultInterrupt));
  } >ivt
```

The vector table is defined as a series of `LONG()` data commands. Each vector table entry is 4 bytes in length (3 bytes for a program memory address plus an unused phantom byte). The data commands include an expression using the `DEFINED()` function and the `?` operator. A typical entry may be interpreted as follows:

If symbol “__OscillatorFail” is defined, insert the absolute address of that symbol. Otherwise, insert the absolute address of symbol “__DefaultInterrupt”.

By convention, a function that will be installed as the second interrupt vector should have the name `__OscillatorFail`. If such a function is included in the link, its address is loaded into the entry. If the function is not included, the address of the default interrupt handler is loaded instead. If the application has not provided a default interrupt handler (i.e., a function with the name `__DefaultInterrupt`), the linker will generate one automatically. The simplest default interrupt handler is a Reset instruction.

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

The contents of the alternate interrupt vector table are defined as follows:

```
/*
** Alternate Interrupt Vector Table
*/
.aivt __AIVT_BASE :
  {
    LONG(DEFINED(__AltReservedTrap0)  ? ABSOLUTE(__AltReservedTrap0)  :
        (DEFINED(__ReservedTrap0)     ? ABSOLUTE(__ReservedTrap0)     :
         ABSOLUTE(__DefaultInterrupt)));
    LONG(DEFINED(__AltOscillatorFail) ? ABSOLUTE(__AltOscillatorFail) :
        (DEFINED(__OscillatorFail)    ? ABSOLUTE(__OscillatorFail)    :
         ABSOLUTE(__DefaultInterrupt)));
    LONG(DEFINED(__AltAddressError)   ? ABSOLUTE(__AltAddressError)   :
        (DEFINED(__AddressError)      ? ABSOLUTE(__AddressError)      :
         ABSOLUTE(__DefaultInterrupt)));
    :
    :
    LONG(DEFINED(__AltInterrupt53)    ? ABSOLUTE(__AltInterrupt53)    :
        (DEFINED(__Interrupt53)       ? ABSOLUTE(__Interrupt53)       :
         ABSOLUTE(__DefaultInterrupt)));
  } >aivt
```

The syntax of the alternate interrupt vector table is similar to the primary, except for an additional expression that causes each alternate table entry to default to the corresponding primary table entry.

10.3.6 SFR Addresses

Absolute addresses for the SFRs are defined as a series of symbol definitions:

```
**=============================================================================
**
**       dsPIC Core Register Definitions
**
**=============================================================================*/
 WREG0 = 0x0000;
_WREG0 = 0x0000;
 WREG1 = 0x0002;
_WREG1 = 0x0002;
:
:
```

Note: If identifiers in a C or assembly program are defined with the same names as SFRs, multiple definition linker errors will result.

Two versions of each SFR address are included, with and without a leading underscore. This is to enable both C and assembly language programmers to refer to the SFR using the same name. By convention, the C compiler adds a leading underscore to every identifier.

10.4 Creating a Custom Linker Script

The standard 16-bit linker scripts are general purpose and will satisfy the demands of most applications. However, occasions may arise where a custom linker script is required.

To create a custom linker script, start with a copy of the standard linker script that is appropriate for the target device. For example, to customize a linker script for the dsPIC30F3014 device, start with a copy of `p30f3014.gld`.

Customizing a standard linker script will usually involve editing sections or commands that are already present. For example, stubs for user-defined sections in both data memory and program memory are included. These stubs may be renamed and/or customized with absolute addresses if required.

It is recommended that unused sections be retained in a custom linker script, since unused sections will not impact application memory usage. If a section must be removed for a custom script, C style comments can be used to disable it.

10.5 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.

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.

10.5.1 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 object file format. Each file is called an object file. Each object file has, among other things, a list of sections. A section in an input file is called an input section; similarly, a section in the output file is an output section.

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

Every loadable or allocatable output section has two addresses. The first is the VMA, or virtual memory address. This is the address the section will have when the output file is run. The second is the LMA, or load memory address. This is the address at which the section will be loaded. In most cases, the two addresses will be the same. An example of when they might be different is when a section is intended for use in the PSV window. In this case, the program memory address would be the LMA, and the data memory address would be the VMA.

The sections in an object file can be viewed by using the `xc16-objdump` program with the `-h` option.

Every object file also has a list of symbols, known as the symbol table. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If a C or C++ program is compiled into an object file, a defined symbol will be created for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

Symbols in an object file can be viewed by using the `xc16-nm` program, or by using the `xc16-objdump` program with the `-t` option.

10.5.2 Commands Dealing with Files

Several linker script commands deal with files.

```
CRT0_STARTUP(object file)
```

This command identifies which primary startup module should be loaded from the compiler libraries. The primary startup module defines reserved symbol `__resetPRI` and is responsible for initializing the C runtime environment. Multiple versions of this module exist in order to support architectural differences between device families. Although the linker expects to find this command in every linker script, a default startup module will be selected if the command is missing (as might be the case with custom linker scripts in legacy projects).

```
CRT1_STARTUP(object file)
```

This command identifies which alternate startup module should be loaded from the compiler libraries. The alternate startup module defines reserved symbol `__resetALT` and is responsible for initializing the C runtime environment without data initialization. Multiple versions of this module exist in order to support architectural differences between device families. Although the linker expects to find this command in every linker script, a default startup module will be selected if the command is missing (as might be the case with custom linker scripts in legacy projects).

```
INCLUDE filename
```

Include the linker script filename at this point. The file will be searched for in the current directory, and in any directory specified with the -L option. Calls to INCLUDE may be nested up to 10 levels deep.

```
INPUT(file, file, ...)
INPUT(file file ...)
```

The INPUT command directs the linker to include the named files in the link, as though they were named on the command line. The linker will first try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of -L in Section 8.4.22 "--library-path <dir> (-L <dir>).".

If INPUT (-lfile) is used, xc16-ld will transform the name to libfile.a, as with the command line argument -l.

When the INPUT command appears in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

```
GROUP(file, file, ...)
GROUP(file file ...)
```

The GROUP command is like INPUT, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of -(in Section 8.4.4 "-(archives -), --start-group archives, --end-group."

```
OPTIONAL(file, file, ...)
OPTIONAL(file file ...)
```

The OPTIONAL command is analogous to the INPUT command, except that the named files are not required for the link to succeed. This is particularly useful for specifying archives (or libraries) that may or may not be installed with the compiler.

```
OUTPUT(filename)
```

The OUTPUT command names the output file. Using OUTPUT(filename) in the linker script is exactly like using -o filename on the command line (see Section 8.4.28 "--output file (-o file)."). If both are used, the command line option takes precedence.

```
SEARCH_DIR(path)
```

The SEARCH_DIR command adds path to the list of paths where the linker looks for archive libraries. Using SEARCH_DIR(path) is exactly like using -L path on the command line (see Section 8.4.22 "--library-path <dir> (-L <dir>)."). If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

```
STARTUP(filename)
```

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

10.5.3 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.

10.5.3.1 Simple Assignments

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

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

```
symbol &= expression ;
symbol |= expression ;
```

The first case will define symbol to the value of expression. In the other cases, symbol must already be defined, and the value will be adjusted accordingly.

The special symbol name '.' indicates the location counter. This symbol may only be used within a `SECTIONS` command.

The semicolon after expression is required.

Expressions are defined in **Section 9.8 Expressions in Linker Scripts**.

Symbol assignments may appear as commands in their own right, or as statements within a `SECTIONS` command, or as part of an output section description in a `SECTIONS` command.

The section of the symbol will be set from the section of the expression; for more information, see Section 9.8.6 "The Section of an Expression."

Here is an example showing the three different places that symbol assignments may be used:

```
  floating_point = 0;
SECTIONS
{
  .text :
  {
    *(.text)
    _etext = .;
  }
  _bdata = (. + 3) & ~ 4;
  .data : { *(.data) }
}
```

In this example, the symbol `floating_point` will be defined as zero. The symbol `_etext` will be defined as the address following the last `.text` input section. The symbol `_bdata` will be defined as the address following the `.text` output section aligned upward to a 4-byte boundary.

10.5.3.2 `PROVIDE` Command

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in the link. For example, traditional linkers defined the symbol `etext`. However, ANSI C requires that `etext` may be used as a function name without encountering an error. The `PROVIDE` keyword may be used to define a symbol, such as `etext`, only if it is referenced but not defined. The syntax is `PROVIDE(symbol = expression)`.

Here is an example of using `PROVIDE` to define `etext`:

```
SECTIONS
{
  .text :
  {
    *(.text)
    _etext = .;
    PROVIDE(etext = .);
  }
}
```

In this example, if the program defines `_etext` (with a leading underscore), the linker will give a multiple definition error. If, on the other hand, the program defines `etext` (with no leading underscore), the linker will silently use the definition in the program. If the program references `etext` but does not define it, the linker will use the definition in the linker script.

10.5.4 `MEMORY` Command

The linker's default configuration permits allocation of all available memory. This can be overridden by using the `MEMORY` command.

The `MEMORY` command describes the location and size of blocks of memory in the target. It can be used to describe which memory regions may be used by the linker and which memory regions it must avoid. Sections may then be

assigned to particular memory regions. The linker will set section addresses based on the memory regions and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

The syntax of the MEMORY command is:

```
MEMORY
{
  name [(attr)] : ORIGIN = origin, LENGTH = len
  ...
}
```

The name is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names or section names. Each memory region must have a distinct name.

The *attr* string is an optional list of attributes associated with the memory region. Historically it was used to determine where unmapped sections should be located by the sequential memory allocator. This capability is no longer used because unmapped sections are now located by the best-fit allocator (for more information see Section 10.5 “Linker Allocation.”).

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.

Note: It is possible to use a preprocessor macro instead of a literal value for the origin and/or length of a memory region.

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.

10.5.5 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 Section 9.7.6 “Other Linker Script Commands.”)
- a symbol assignment (see Section 9.7.3 “Assigning Values to Symbols.”)
- an output section description
- an overlay description

The ENTRY command and symbol assignments are permitted inside the SECTIONS command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because those commands can be used at meaningful points in the layout of the output file.

Output section descriptions and overlay descriptions are described below.

If a SECTIONS command does not appear in the linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are

present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

10.5.5.1 Input Section Description

The most common output section command is an input section description.

The input section description is the most basic linker script operation. Output sections tell the linker how to lay out the program in memory. Input section descriptions tell the linker how to map the input files into the memory layout.

An input section description consists of a file name optionally followed by a list of section names in parentheses.

The file name and the section name may be wildcard patterns, which are described further below.

The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input `.text` sections, one would write:

```
*(.text)
```

Here the `*` is a wildcard which matches any file name. To exclude a list of files from matching the file name wildcard, `EXCLUDE_FILE` may be used to match all files except the ones specified in the `EXCLUDE_FILE` list. For example:

```
*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```

will cause all `.ctors` sections from all files except `crtend.o` and `otherfile.o` to be included.

There are two ways to include more than one section:

```
*(.text .rdata)
*(.text) *(.rdata)
```

The difference between these is the order in which the `.text` and `.rdata` input sections will appear in the output section. In the first example, they will be intermingled. In the second example, all `.text` input sections will appear first, followed by all `.rdata` input sections.

A file name can be specified to include sections from a particular file. This would be useful if one of the files contain special data that needs to be at a particular location in memory. For example:

```
data.o(.data)
```

If a file name is specified without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may be useful on occasion. For example:

```
data.o
```

When a file name is specified which does not contain any wild card characters, the linker will first see if the file name was also specified on the linker command line or in an `INPUT` command. If not, the linker will attempt to open the file as an input file, as though it appeared on the command line. This differs from an `INPUT` command because the linker will not search for the file in the archive search path.

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

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

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

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

10.5.5.3 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.

10.5.5.4 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)
  }
}
```

10.5.5.5 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 use most of the optional section attributes.

The white space around *name* and *address* is required. The colon and the curly braces are also required. The line breaks and other white space are optional.

A section name may consist of any sequence of characters, but a name which contains any unusual characters such as commas must be quoted.

Each output-section-command may be one of the following:

- a symbol assignment (see Section 9.7.3 “Assigning Values to Symbols.”)
- an input section description (see Section 9.7.5.1 “Input Section Description.”)
- data values to include directly (see Section 9.7.5.7 “Output Section Data.”)

10.5.5.6 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 (**Section 9.8 Expressions in Linker Scripts**). For example, to align the section on a `0x10` byte boundary, so that the lowest four bits of the section address are zero, the command could look like this:

```
.text ALIGN(0x10) : { *(.text) }
```

This works because `ALIGN` returns the current location counter aligned upward to the specified value.

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

10.5.5.7 Output Section Data

Explicit bytes of data may be inserted into an output section by using `BYTE`, `SHORT`, or `LONG` 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`, or `LONG` commands store one, two, or four bytes (respectively). For example, this command will store the four byte value of the symbol `addr`:

```
LONG(addr)
```

After storing the bytes, the location counter is incremented by the number of bytes stored. When using data commands in a program memory section, it is important to note that the linker considers program memory to be 32-bits wide, even though only 24 bits are physically implemented. Therefore, the most significant 8 bits of a `LONG` data value are not loaded into device memory.

Data commands only work inside a section description and not between them, so the following will produce an error from the linker:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

whereas this will work:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

The `FILL` command may be used to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the two least significant bytes of the expression, repeated as necessary. A `FILL` statement covers memory locations after the point at which it occurs in the section definition; by including more than one `FILL` statement, different fill patterns may be used in different parts of an output section.

This example shows how to fill unspecified regions of memory with the value `0x9090`:

```
FILL(0x9090)
```

The `FILL` command is similar to the `=fillexp` output section attribute (see Section 9.7.5.9 “Output Section Attributes.”), but it only affects the part of the section following the `FILL` command, rather than the entire section. If both are used, the `FILL` command takes precedence.

10.5.5.8 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.

10.5.5.9 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]
```

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

10.5.5.10 Output Section Type

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

NOLOAD

The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.

DSECT, COPY, INFO, OVERLAY

These type names are supported for backward compatibility, and are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.

The linker normally sets the attributes of an output section based on the input sections which map into it. This can be overridden by using the section type. For example, in the script sample below, the `ROM` section is addressed at memory location `0` and does not need to be loaded when the program is run. The contents of the `ROM` section will appear in the linker output file as usual.

```
SECTIONS {
 ROM 0 (NOLOAD) : { ... }
 ...
}
```

10.5.5.11 Output Section LMA

Every section has a virtual address (VMA) and a load address (LMA). The address expression which may appear in an output section description sets the VMA.

The linker will normally set the LMA equal to the VMA. This can be changed by using the `AT` keyword.
The expression lma that follows the `AT` keyword specifies the load address of the section. Alternatively, with `AT>lma_region` expression, a memory region may be specified for the section's load address (see Section 9.7.4 "`MEMORY` Command.").

This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called `.text`, which starts at `0x1000`, one called `.mdata`, which is loaded at the end of the `.text` section even though its VMA is `0x2000`, and one called `.bss` to hold uninitialized data at address `0x3000`. The `symbol _data` is defined with the value `0x2000`, which shows that the location counter holds the VMA value, not the LMA value.

```
SECTIONS
{
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = . ; *(.data); _edata = . ;  }
  .bss 0x3000 :
    { _bstart = . ;  *(.bss) *(COMMON) ; _bend = . ;}
}
```

The run-time initialization code for use with a program generated with this linker script would include a function to copy the initialized data from the ROM image to its run-time address. The initialization function could take advantage of the symbols defined by the linker script.

It would rarely be necessary to write such a function, however. The 16-bit linker includes automatic support for the initialization of BSS-type and data-type sections. Instead of mapping a data section into both program memory and data memory (as this example implies), the linker creates a special template in program memory which includes all of the relevant information. See Section 10.8 "Initialized Data." for details.

10.5.5.12 Output Section Region

A section can be assigned to a previously defined region of memory by using `>region`. See Section 9.7.4 "`MEMORY` Command."

Here is a simple example:

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

10.5.5.13 Output Section Fill

A fill pattern can be set for an entire section by using `=fillexp`. `fillexp` as an expression. Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the two least significant bytes of the value, repeated as necessary.

The fill value can also be changed with a `FILL` command in the output section commands; see Section 9.7.5.7 “Output Section Data.”

Here is a simple example:

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

10.5.5.14 Overlay Description

An overlay description provides an easy way to describe sections which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the run-time memory address as required, perhaps by simply manipulating addressing bits.

This approach is not suitable for defining sections that will be used with the PSV window, because the `OVERLAY` command does not permit individual load addresses to be specified for each section. Instead, the 16-bit linker provides automatic support for read-only sections in the PSV window. See Section 10.9 “Read-only Data.” for details.

Overlays are described using the `OVERLAY` command. The `OVERLAY` command is used within a `SECTIONS` command, like an output section description. The full syntax of the `OVERLAY` command is as follows:

```
OVERLAY [start] : [NOCROSSREFS] [AT ( ldaddr )]
{
  secname1
  {
    output-section-command
    output-section-command
    ...
  } [:phdr...] [=fill]
  secname2
  {
    output-section-command
    output-section-command
    ...
  } [:phdr...] [=fill]
  ...
} [>region] [:phdr...] [=fill]
```

Everything is optional except `OVERLAY` (a keyword), and each section must have a name (`secname1` and `secname2` above). The section definitions within the `OVERLAY` construct are identical to those within the general `SECTIONS` construct, except that no addresses and no memory regions may be defined for sections within an `OVERLAY`.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the `OVERLAY` as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to the current value of the location counter).

If the `NOCROSSREFS` keyword is used, and there are any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another.

For each section within the `OVERLAY`, the linker automatically defines two symbols. The symbol `__load_start_secname` is defined as the starting load address of the section. The symbol `__load_stop_secname` is defined as the final load address of the section. Any characters within secname which are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary.

At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section.

Here is an example. Remember that this would appear inside a SECTIONS construct.

```
OVERLAY 0x1000 : AT (0x4000)
{
 .text0 { o1/*.o(.text) }
 .text1 { o2/*.o(.text) }
}
```

This will define both .text0 and .text1 to start at address 0x1000. .text0 will be loaded at address 0x4000, and .text1 will be loaded immediately after .text0. The following symbols will be defined: __load_start_text0, __load_stop_text0, __load_start_text1, __load_stop_text1.

C code to copy overlay .text1 into the overlay area might look like the following:

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1, &__load_stop_text1 - &__load_start_text1);
```

The OVERLAY command is a convenience, since everything it does can be done using the more basic commands. The previous example could have been written identically as follows.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

10.5.6 Other Linker Script Commands

There are several other linker script commands, which are described briefly:

ENTRY(*symbol*)

Specify *symbol* as the first instruction to execute in the program. The linker will record the address of this symbol in the output object file header. This does not affect the Reset instruction at address zero, which must be generated in some other way. By convention, the 16-bit linker scripts construct a GOTO __reset instruction at address zero.

EXTERN(*symbol symbol* ...)

Force *symbol* to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. Several symbols may be listed for each EXTERN, and EXTERN may appear multiple times. This command has the same effect as the -u command line option.

FORCE_COMMON_ALLOCATION

This command has the same effect as the -d command line option: to make 16-bit linker assign space to common symbols even if a relocatable output file is specified (-r).

NOCROSSREFS(*section section* ...)

This command may be used to tell 16-bit linker to issue an error about any references among certain output sections. In certain types of programs, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors.

The NOCROSSREFS command takes a list of output section names. If the linker detects any cross references between the sections, it reports an error and returns a non-zero exit status. The NOCROSSREFS command uses output section names, not input section names.

OUTPUT_ARCH(*processor_name*)

Specify a target processor for the link. This command has the same effect as the `-p,--processor` command line option. If both are specified, the command line option takes precedence. The processor name should appear in quotes; for example “30F6014”, “24FJ128GA010”, or “33FJ128GP706”.

```
OUTPUT_FORMAT(format_name)
```

The `OUTPUT_FORMAT` command names the object file format to use for the output file.

```
TARGET(bfdname)
```

The `TARGET` command names the object file format to use when reading input files. It affects subsequent `INPUT` and `GROUP` commands.

10.6 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.

10.6.1 Constants

All constants are integers.

As in C, the linker considers an integer beginning with `O` 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;
```

10.6.2 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.

10.6.3 The Location Counter

The special linker variable dot ‘`.`’ always contains the current output location counter. Since the ‘`.`’. always refers to a location in an output section, it may only appear in an expression within a `SECTIONS` command. The ‘`.`’ symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to ‘`.`’ will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
{
  output :
  {
    file1(.text)
    . = . + 1000;
    file2(.text)
```

```
    . += 1000;
    file3(.text)
  } = 0x1234;
}
```

In the previous example, the `.text` section from `file1` is located at the beginning of the output section `output`. It is followed by a 1000 byte gap. Then the `.text` section from `file2` appears, also with a 1000 byte gap following before the `.text` section from `file3`. The notation `= 0x1234` specifies what data to write in the gaps.

'`.`' actually refers to the byte offset from the start of the current containing object. Normally this is the `SECTIONS` statement, whose start address is `0`, hence '`.`' can be used as an absolute address. If '`.`' is used inside a section description, however, it refers to the byte offset from the start of that section, not an absolute address, as shown in the following script:

```
SECTIONS
{
  . = 0x100
  .text: {
    *(.text)
    . = 0x200
  }
  . = 0x500
  .data: {
    *(.data)
    . += 0x600
  }
}
```

The `.text` section will be assigned a starting address of `0x100` and a size of exactly `0x200` bytes, even if there is not enough data in the `.text` input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move '.' backwards). The `.data` section will start at `0x500` and it will have an extra `0x600` bytes worth of space after the end of the values from the `.data` input sections and before the end of the `.data` output section itself.

10.6.4 Operators

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

Table 10-2. 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

10.6.5 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".

10.6.6 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.

10.6.7 Built-in Functions

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

10.6.7.1 ABSOLUTE(*exp*)

Return the absolute (non-relocatable, as opposed to non-negative) value of the expression *exp*. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative (see Section 9.8.6 "The Section of an Expression.").

10.6.7.2 ADDR(*section*)

Return the absolute address (the VMA) of the named section. Your script must previously have defined the location of that section. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

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

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

10.6.7.3 ALIGN(*exp*)

Return the location counter (`.`) aligned to the next exp boundary. *exp* must be an expression whose value is a power of two. This is equivalent to:

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

`ALIGN` doesn't change the value of the location counter; it just does arithmetic on it. Here is an example which aligns the output `.data` section to the next `0x2000` byte boundary after the preceding section and sets a variable within the section to the next `0x8000` boundary after the input sections:

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
  ...
}
```

The first use of `ALIGN` in this example specifies the location of a section because it is used as the optional address attribute of a section definition, see Section 9.7.5 "`SECTIONS` Command." The second use of `ALIGN` is used to define the value of a symbol.

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

10.6.7.4 ASSERT(*exp, message*)

Ensure that exp is non-zero. If it is zero, then exit the linker with an error code, and print *message*. E.g.,

```
__CHECK = ASSERT(1, "OK");
```

10.6.7.5 BLOCK(*exp*)

This is a synonym for `ALIGN`, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

10.6.7.6 DEFINED(*symbol*)

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

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

10.6.7.7 LOADADDR(*section*)

Return the absolute LMA of the named section. This is normally the same as `ADDR`, but it may be different if the AT attribute is used in the output section definition, see Section 9.7.5 "`SECTIONS` Command."

10.6.7.8 MAX(*exp1, exp2*)

Returns the maximum of *exp1* and *exp2*.

10.6.7.9 MIN(*exp1, exp2*)

Returns the minimum of *exp1* and *exp2*.

10.6.7.10 NEXT(*exp*)

Return the next unallocated address that is a multiple of *exp*. This function is equivalent to `ALIGN(exp)`.

10.6.7.11 SIZEOF(*section*)

Return the size in bytes of the named section, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. In the following example, `symbol_1` and `symbol_2` are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
  }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
  ...
}
```

11. Linker Processing

How the MPLAB XC16 Object Linker builds an application from input files and the linker script is discussed here.

11.1 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 6 steps, discussed in the following sections.

11.1.1 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 and SFR addresses. See 10. 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.

11.1.2 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 11.3. Linker Allocation

11.1.3 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 11.4. Global and Weak Symbols. If any external symbol reference remains undefined, an error is generated.

References to redundant functions in archive files will be merged in order to conserve memory. For example, both integer and floating-point versions of the standard C formatted I/O functions are included in `libc.a`. The 16-bit compiler will generate references to the appropriate function, based on a static analysis of format strings. When multiple object files are combined by the linker, both versions of a particular I/O function may be referenced. In such cases the integer functions are redundant, since they represent a subset of the floating-point functionality. The linker will detect this situation, and merge the I/O functions together to conserve memory. This optimization may be disabled with the `--no-smart-io` option.

11.1.4 Creating Special Sections

After the symbols have been resolved, the linker constructs any special input or output sections that are required. For example, the compiler or assembler may have created function pointers using the `handle()` operator. The linker then builds a special input section named `.handle` to implement a jump table. For more information about handles, see 11.5. Handles.

The linker also constructs a special input section named `.dinit` to support initialized data. Section `.dinit` is an initialization template that is interpreted by the C run-time library. For more information about initialized data, see 11.6. Initialized Data.

If the application has not defined a default interrupt handler, the linker will create one automatically in a special input section named `.isr`. Unused slots in the interrupt vector tables are populated with the address of this function. For more information on the default interrupt handler, see section 11.10. Interrupt Vector Tables [DD].

11.1.5 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.

Boundaries of the stack and heap are calculated, based on the extent of unused data memory. If insufficient memory is available, an error is generated. For more information about the stack and heap, see 11.8. Stack Allocation and 11.9. Heap Allocation.

11.1.6 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, if a function pointer is referenced without the `handle()` operator and its address is too large to fit in 16 bits.

A link map is also generated if requested with the appropriate option. The link map includes a memory usage report, which shows the starting address and length of all sections in data memory and program memory. For more information about the link map, see 10.3.4. Input/Output Section Map.

11.2 Memory Addressing

The 16-bit devices use a modified Harvard architecture with separate data and program memory spaces. Data memory is both byte-oriented (8 bits wide) and word-oriented (16 bits wide). Bytes are assigned sequential addresses, starting with 0, 1, 2, 3 and so on. Words are assigned sequential even addresses, starting with 0, 2, 4, 6 and so on.

Program memory is word-oriented, where each instruction word is 24 bits wide. Instruction words are assigned sequential even addresses, starting with 0, 2, 4, 6 and so on. The PC indicates the next instruction to be executed, and increments by 2 for each instruction word. Individual bytes in a program memory word are not addressable.

While a traditional Harvard architecture does not permit access to data stored in program memory, the 16-bit architecture provides three ways to accomplish this task, discussed in the following sections.

11.2.1 Table Access Instructions

The table access instructions `tblrdl`, `tblrdh`, `tblwtl` and `tblwth` can be used to access data stored in program memory. Data is addressed through a 16-bit data register pointer in combination with the 8-bit TBLPAG register. The special operators `tbloffset()` and `tblpage()` facilitate table access in assembly language. See the 16-bit assembler documentation, “**Table Read/Write Instructions**”, for more information .

The linker resolves symbolic references to labels in program memory for use with the table access instructions. Although data in program memory can be specified one byte at a time, only the least-significant byte in each instruction word has a unique address. For example, consider the following assembly source code example:

```
.section prog,code
L1:  .pbyte 1
```

```
L2:  .pbyte 2
L3:  .pbyte 3
L4:  .pbyte 4
.pbyte 5
.pbyte 6
.pbyte 7,8,9
```

In this example, the `code` section attribute designates a section to be allocated in program memory, and the `.pbyte` directives define individual byte constants. Since labels must resolve to a valid PC address, the assembler adds padding after each of the first three constants. Subsequent constants do not require padding. The following assembly listing excerpt illustrates the organization of these constants in program memory:

```
1                       .section prog,code
2 000000  01 00 00      L1:.pbyte 1
3 000002  02 00 00      L2:.pbyte 2
4 000004  03 00 00      L3:.pbyte 3
5 000006  04            L4:.pbyte 4
6            05         .pbyte 5
7               06      .pbyte 6
8 000008  07 08 09      .pbyte 7,8,9
```

Constants 1, 2, 3 are padded out to a full instruction word and have unique PC addresses. Constants 4, 5, 6 are packed into a single instruction word and share the same address.

11.2.2 Program Space Visibility (PSV) Window

The Program Space Visibility window can be used to access data stored in the least significant 16 bits of program memory. When PSV is enabled, the upper 32K of data memory space (0x8000-0xFFFF) functions as a window into program memory. Data is addressed through a 16-bit data register pointer in combination with the 8-bit PSVPAG register. The special operators `psvoffset()` and `psvpage()` are provided to facilitate PSV access in assembly language. Built-in functions `__builtin_psvoffset()` and `__builtin_psvpage()` are provided to facilitate PSV access in C.

The linker supports PSV window operations through the use of read-only data sections. For a detailed discussion of read-only sections, see 11.7. Read-only Data.

11.2.3 Extended Data Space (EDS) Window

Some device families support a new data memory architecture called Extended Data Space (EDS). EDS extends the functionality of the PSV window to access additional pages of RAM as well as memory-mapped peripherals. On an EDS device, the PSVPAG register has been replaced by two registers:

- DSRPAG for reading from Flash, RAM, etc.
- DSWPAG for writing to RAM

The operation of the EDS window is analogous to the PSV window. When the page registers are set appropriately, a portion of program memory (or extended data memory) can be accessed in the data address range 0x8000 to 0xFFFF. Unlike the PSV window, the EDS window is always enabled. Another difference is that certain page number ranges imply different address spaces:

EDS Page Range	Description
0x001 to 0x1FF	read/write access to RAM
0x200 to 0x2FF	read-only access to lower 16 bits of program memory
0x300 to 0x3FF	read-only access to upper 8 bits of program memory

Note: EDS page 0 is undefined. Application code should not attempt to access the EDS window with a page value of zero. Such access is prohibited and a hardware exception will occur.

The special operators `edsoffset()` and `edspage()` are provided to facilitate EDS access from assembly language. Built-in functions `__builtin_edsoffset()` and `__builtin_edspage()` are provided to facilitate EDS access from C.

The EDS special operators may be used to access any object in on-chip memory, including local RAM (i.e., RAM located within the first 32K of data address space). Consequently, `edsoffset()` may return a pointer in the range

0x0 to 0xFFFF. `edspage()` will return a page value in the range 0x001 to 0x2FF. Page values greater that 0x300 are not currently supported.

11.3 Linker Allocation

Linker allocation is controlled by the linker script, and proceeds in three steps:

1. Mapping Input Sections to Output Sections
2. Assigning Output Sections to Regions
3. Allocating Unmapped Sections

Steps 1 and 2 are performed by a sequential memory allocator. Input sections which appear in the linker script are assigned to specific memory regions in the target devices. Addresses within a memory region are allocated sequentially, beginning with the lowest address and growing upwards.

Step 3 is performed by a best-fit memory allocator. Input sections which do not appear in the linker script are assigned to memory regions according to their attributes. The best-fit allocator makes efficient use of any remaining memory, including gaps between output sections that may have been left by the sequential allocator.

If memory has been reserved for the `boot` and/or `secure` segments, it will be allocated by the best-fit allocator in step 3. The sequential allocator will avoid these segments, so sections designated with the `boot` or `secure` attributes should not appear in the linker script.

11.3.1 Mapping Input Sections to Output Sections

Input sections are grouped and mapped into output sections, according to the section map. When an output section contains several different input sections, the exact ordering of input sections may be important. For example, consider the following output section definition:

```
/*
** User Code and Library Code
*/
.text :
{
*(.init);
*(.user_init);
*(.handle);
*(.libc) *(.libm) *(.libdsp); /* keep together in this order */
*(.lib*);
} >program
```

Here the output section named `.text` is defined. Notice that the contents of this section are specified within curly braces {}. After the closing brace, `>program` indicates that this output section should be assigned to memory region program.

The contents of output section `.text` may be interpreted as follows:

- Input sections named `.init` are collected and mapped into the output section. By convention, there is only one `.init` section, and it contains the startup code for an application. It appears first in the output section (i.e., at the lowest address) so that its address is readily available if necessary.
- Input sections named `.user_init` are collected and mapped into the output section. These sections are created by the compiler and refer to functions that have been decorated with the `user_init` attribute. Their position within the output section is not critical, but since they are associated with section.init, they are located immediately after.
- All input sections named `.handle` are collected and mapped into the output section. `.handle` sections occupy a relatively low address range, which is a requirement for code handles.
- Input sections named `.libc`, `.libm` and `.libdsp` are collected and mapped into the output section. Grouping these sections ensures locality of reference for the run-time library functions, so that PC-relative instructions can be used for maximum efficiency.
- Input sections which match the wildcard pattern `.lib*` are collected and mapped into the output section. This includes libraries such as the peripheral libraries (which are allocated in section `.libperi`).

11.3.2 Assigning Output Sections to Regions

Once the sizes of all output sections are known, they are assigned to memory regions. Normally a region is specified in the output section definition. If a region is not specified, the first defined memory region will be used.

Memory regions are filled sequentially, from lower to higher addresses, in the same order that sections appear in the section map. Memory reserved for `boot` or `secure` segments will be avoided, as well as sections that have been marked with the `address` attribute in source code. A location counter, unique to each region, keeps track of the next available memory location. There are two conditions which may cause gaps in the allocation of memory within a region:

- The section map specifies an absolute address for an output section, or
- The output section has a particular alignment requirement.

In either case, any intervening memory between the current location counter and the absolute (or aligned) address is skipped. Once a range of memory has been skipped, it is available for use by the best-fit allocator. The exact address of all items allocated in memory may be determined from the link map file.

Section alignment requirements typically arise in DSP programming. To utilize modulo addressing, it is necessary to align a block of memory to a particular storage boundary. This can be accomplished with the `aligned` attribute in C, or with the `.align` directive in assembly language. The section containing an aligned memory block must also be aligned, to the same (or greater) power of 2. If two or more input sections have different alignment requirements, the largest alignment is used for the output section.

Another restriction on memory allocation is associated with read-only data sections. Read-only data sections are identified with the `psv` section attribute and are dedicated for use in the PSV window or the Extended Data Space (EDS) window. The C compiler creates a read-only data section named `.const` to store constants when the `--mconst-in-code` option is selected.

To allow efficient access of constant tables in the PSV or EDS window, the linker ensures that a read-only section will not cross a page boundary. Therefore a single setting of the page register can be used to access the entire section. If necessary, output sections in program memory will be re-sorted after the sequential allocation pass to accommodate this restriction. If an absolute address has been specified in the linker script for a particular section, it will not be moved. In general, fully relocatable sections provide the most flexibility for efficient memory allocation.

Note: Sections with specific alignment requirements, such as `psv` sections or sections intended for modulo addressing, may be allocated most efficiently by the best-fit allocator. For best-fit allocation, these sections should not appear in the linker script.

11.3.3 Allocating Unmapped Sections

After all sections that appear in the section map are allocated, any remaining sections are considered to be unmapped. Unmapped sections are allocated according to section attributes. The linker uses a best-fit memory allocator to determine the most efficient arrangement in memory. The primary emphasis of the best-fit allocator is the reduction or elimination of memory gaps due to address alignment restrictions.

Since data memory is limited on many 16-bit devices, and several architectural features imply address alignment restrictions, efficient allocation of data memory is particularly important. By convention, data memory sections are not explicitly mapped in linker scripts, thus providing maximum flexibility for the best-fit memory allocator.

Section attributes affect memory allocation as described below. For a general discussion of section attributes, see 5.1. Directives that Define Sections.

11.3.3.1 code

The code attribute specifies that a section should be allocated in program memory, as defined by region `program` in the linker script. The following attributes may be used in conjunction with code and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `boot` specifies the boot segment
- `secure` specifies the secure segment

11.3.3.2 data [DD]

The data attribute specifies that a section should be allocated as initialized storage in data memory, as defined by region `data` in the linker script. The following attributes may be used in conjunction with data and will further specify the allocation:

- `address()` specifies an absolute address
- `near` specifies the first 8K of data memory
- `xmemory` [DD] specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `ymemory` [DD] specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `dma` [DD] specifies dma address space, which includes the portion of region data between addresses __DMA_BASE and __DMA_END as defined in the linker script (for PIC24H MCUs and dsPIC33F DSCs only).

11.3.3.3 bss [DD]

The bss attribute specifies that a section should be allocated as uninitialized storage in data memory, as defined by region `data` in the linker script. The following attributes may be used in conjunction with bss and will further specify the allocation:

- `address()` specifies an absolute address
- `near` specifies the first 8K of data memory
- `xmemory` [DD] specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `ymemory` [DD] specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `dma` [DD] specifies dma address space, which includes the portion of region data between addresses __DMA_BASE and __DMA_END as defined in the linker script (for PIC24H MCUs and dsPIC33F DSCs only).
- `boot` specifies the boot segment
- `secure` specifies the secure segment

11.3.3.4 persist [DD]

The persist attribute specifies that a section should be allocated as persistent storage in data memory, as defined by region `data` in the linker script. Persistent storage is not cleared or initialized by the C run-time library. The following attributes may be used in conjunction with persist and will further specify the allocation:

- `address()` specifies an absolute address
- `near` specifies the first 8K of data memory
- `xmemory` [DD] specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `ymemory` [DD] specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `dma` [DD] specifies dma address space, which includes the portion of region data between addresses __DMA_BASE and __DMA_END as defined in the linker script (for PIC24H MCUs and dsPIC33F DSCs only).

11.3.3.5 psv

The psv attribute specifies that a section should be allocated in program memory, as defined by region `program` in the linker script. psv sections are intended for use with the Program Space Visibility window or the Extended Data Space (EDS) window, and will be located so that the entire contents may be accessed using a single setting of the page register. This allocation rule implies that the total size of a psv section can not exceed 32K. The following attributes may be used in conjunction with psv and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `boot` specifies the boot segment
- `secure` specifies the secure segment

11.3.3.6 memory

The memory attribute specifies that a section should be allocated in external or user-defined memory. The following attributes may be used in conjunction with memory and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `noload` specifies that the section should not be loaded with the primary application
  Note: Sections allocated in external or user-defined memory cannot be accessed by the PSV window or the EDS window.

eedata – dsPIC30F DSCs only [DD]

The eedata attribute specifies that a section should be allocated in data EEPROM memory, as defined by region `eedata` in the linker script. The following attributes may be used in conjunction with eedata and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address
- `reverse()` specifies alignment of the section ending address + 1
- `boot` specifies the boot segment
- `secure` specifies the secure segment

11.3.3.7 heap [DD]

The heap attribute specifies that a section should be designated for use by the C run-time library for dynamic memory allocation. The heap must always be allocated in local data memory (address range 0x0 to 0x7FFE). The following attributes may be used in conjunction with heap and will further specify the allocation:

- `address()` specifies an absolute address
- `xmemory` [DD] specifies X address space, which includes all of region data below the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `ymemory` [DD] specifies Y address space, which includes all of region data above the address __YDATA_BASE as defined in the linker script (dsPIC30F/33F DSCs only)
- `align()` specifies alignment of the section starting address

11.3.3.8 stack

The stack attribute specifies that a section should be designated for use as the processor stack. On most devices, the stack must always be allocated in local data memory (address range 0x0 to 0x7FFE). On some devices, the stack may be located anywhere in EDS page 1 (address range 0x0 to 0xFFFE). The following attributes may be used in conjunction with stack and will further specify the allocation:

- `address()` specifies an absolute address
- `align()` specifies alignment of the section starting address

11.4 Global and Weak Symbols

When a symbol reference appears in an object file without a corresponding definition, the symbol is declared external. By default, external symbols have global binding and are referred to as global symbols. External symbols may be explicitly declared with weak binding, using the `__weak__` attribute in C or the `.weak` directive in assembly language.

As the name implies, global symbols are visible to all input files involved in the link. There must be one (and only one) definition for every global symbol referenced. If a global definition is not found among the input files, archives will be searched and the first archive module found that contains the needed definition will be loaded. If no definition is found for a global symbol a link error is reported.

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

11.5 Handles

The modified Harvard architecture of dsPIC30F devices supports two memory spaces of unequal size. Data memory space can be fully addressed with 16 bits while program memory space requires 24 bits. Since the native integer data type (register width) is only 16 bits, there is an inherent difficulty in the allocation and manipulation of function pointers that require a full 24 bits. Reserving a pair of 16-bit registers to represent every function pointer is inefficient in terms of code space and execution speed, since many programs will fit in 64K words of program space or less. However, the linker must accommodate function pointers throughout the full 24-bit range of addressable program memory.

Note: Future versions of the compiler may define function pointers to be 24 bits or larger. In such cases, handles will not be used.

In order to ensure a valid 16-bit pointer for any function in the full program memory address space, the 16-bit assembler and linker support the `handle()` operator. The C compiler uses this operator whenever a function address is taken. Assembly programmers can use this operator three different ways:

```
mov    #handle(func),w0 ; handle() used in an instruction
.word  handle(func)     ; handle() used with a data word directive
.pword handle(func)     ; handle() used with a instruction word
;directive
```

The linker searches all input files for handle operators and constructs a jump table in a section named `.handle`. For each function that is referenced by one or more handle operators, a single entry is made in the jump table. Each entry is a `GOTO` instruction. Note that `GOTO` is capable of reaching any function in the full 24- bit address space. Section `.handle` is allocated low in program memory, well within the range of a 16-bit pointer.

When the output file is built, the absolute addresses of all functions are known. Each handle relocation entry is filled with an absolute address. If the address of the target function fits in 16 bits, it is inserted directly into the object code. If the absolute address of the target function exceeds 16 bits, the address of the corresponding entry in the jump table is used instead. Only functions located beyond the range of 16-bit addressing suffer any performance penalty with this technique. However, there is a code space penalty for each unused entry in the jump table.

In order to conserve program memory, the handle jump table can be suppressed for certain devices, or whenever the application programmer is sure that all function pointers will fit in 16 bits. One way is to specify the `--no-handles` link option on the command line or in the IDE. Another way is to define a symbol named `__NO_HANDLES` in the linker script:

```
__NO_HANDLES = 1;
```

Linker scripts for 16-bit devices with 32K instruction words or less all contain the `__NO_HANDLES` definition to suppress the handle jump table.

Note: If the handle jump table is suppressed, and the target address of a function pointer does not fit in 16 bits, a “relocation truncated” link error will be generated.

11.6 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.

11.6.1 Standard Data Section Names

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

Table 11-1. 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` 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 16-bit compiler will automatically define sections to contain variables and functions as needed. For more information on the attributes of variables and functions that may result in automatic section definition, see the “*MPLAB XC16 C Compiler User's Guide” (DS50002071)*.

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.

11.6.2 Data Initialization Template

As noted in 11.6.1. Standard Data Section Names, the 16-bit Language Tools support BSS-type sections (memory that is not initialized) as well as data-type sections (memory that receives initial values). The data-type sections receive initial values at start-up and the BSS-type sections are filled with zeros.

A generic data initialization template is used that supports any number of arbitrary BSS-type sections or data-type sections. The data initialization template is created by the linker and is loaded into an output section named `.dinit` in program memory. Start-up code in the run-time library interprets the template and initializes data memory accordingly.

The data initialization template contains one record for each output section in data memory. The template is terminated by a null instruction word. The format of a data initialization record is:

```
/* data init record */
struct data_record {
char         *dst;      /* destination address  */
unsigned int len;       /* length in bytes      */
unsigned int format:7;  /* format code          */
unsigned int page:9;    /* destination page     */
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 fourth element is the page value of the section. On EDS devices, the page value will be in the range 0x001 to 0x1FF. On all other devices, the page value will be zero. The last element is an optional array of data bytes. For BSS-type sections, no data bytes are required.

The format code has three possible values.

Table 11-2. Format Code Values

Format Code	Description
0	Fill the output section with zeros
1	Copy 2 bytes of data from each instruction word in the data array
2	Copy 3 bytes of data from each instruction word in the data array

By default, data records are created using format 2. Format 2 conserves program memory by using the entire 24-bit instruction word to store initial values. Note that this format causes the encoded instruction words to appear as random and possibly invalid instructions if viewed in the disassembler.

Format 1 data records may be created by specifying the `--no-pack-data` option. Format 1 uses only the lower 16 bits of each 24-bit instruction word to store initial values. The upper byte of each instruction word is filled with 0x0 by default and causes the template to appear as `NOP` instructions if viewed in the disassembler (and will be executed as such by the 16-bit device). A different value may be specified for the upper byte of the data template with the `--fill-data` option.

11.6.3 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 `libpic30-coff.a` or `libpic30-elf.a`.

Multiple versions of the start-up modules are contained within the runtime library. The linker will select a startup module based on commands in the linker script. For example:

```
CRT0_STARTUP(crt0_standard.o)
CRT1_STARTUP(crt1_standard.o)
```

For each device, two start-up modules are specified: a primary module (CRT0) and an alternate module (CRT1).

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 11-3. Main Entry Points

Main Entry Name	Description
`__reset`	Takes control immediately after device Reset
`_main`	Takes control after the start-up module completes its work

Note that the first entry name (`__reset`) includes two leading underscore characters. The second entry name (`_main`) includes only one leading underscore character. The linker scripts construct a `GOTO __reset` instruction at location 0 in program memory, which transfers control upon device Reset.

The primary start-up module is linked by default and performs the following:

1. The stack pointer (W15) and stack pointer limit register (SPLIM) are initialized, using values provided by the linker or a custom linker script. For more information, see 11.8. Stack Allocation.
2. If a `.const` section is defined, it is mapped into the PSV window by initializing the PSVPAG and CORCON registers. On devices which support EDS the DSRPAG register will be initialized. Note that a `.const` section is defined when the "Constants in code space" option is selected in MPLAB IDE, or the `-mconst-in-code` option is specified on the compiler command line.
3. The data initialization template in section `.dinit` is read, causing all uninitialized sections to be cleared, and all initialized sections to be initialized with values read from program memory.
4. If the application has defined `user_init` functions, section `.user_init` is called.

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

   The alternate start-up module is linked when the `--no-data-init option` is specified. It performs the same operations, except for step (3), which is omitted. The alternate start-up module is much smaller than the primary module, and can be selected to conserve program memory if data initialization is not required.

   Source code for both modules is provided in the `src` directory of the MPLAB XC16 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.

11.7 Read-only Data

Read-only data sections are located in program memory, but are defined and accessed just like data memory. They are useful for storing constant tables that are too large for available data memory. The C compiler creates a read-only section named `.const` when the `-mconst-in-code` option is specified.

Access to read-only data sections is provided by means of the PSV window, or the EDS window. In either case, a reference to the read-only data is resolved to a data address within the PSV or EDS window.

C programmers can use the `space` attribute to allocate variables in read-only data sections. Access to such variables can be managed automatically by the compiler, or by explicit application code. For additional information on using read-only variables in C, refer to "*MPLAB® XC16 C Compiler User's Guide*" (DS50002071), Section 4.14 "Program Space Visibility (PSV) Usage" and Section 6.2 "Managed PSV Pointers".

The `psv` section attribute is used to designate read-only data sections in assembly language. The contents of read-only data sections may be specified with data directives, as shown in the following assembly source example:

```
.section rdonly,psv
L1:  .byte 1
L2:  .byte 2
```

In this example, section `rdonly` will be allocated in program memory. Both byte constants will be located in the same program memory word, followed by a pad byte. Unlike other sections in program memory, read-only sections are byte addressable. Each label is resolved to a unique address that lies with the PSV or EDS address range.

The linker allocates read-only sections such that they do not cross a page boundary. Therefore, a single setting of the page register will access the entire section. A maximum length restriction is implied; the linker will issue an error message if any read-only data section exceeds 32 Kbytes. Only the least significant 16 bits of each instruction word are available for data storage (bits 16-23). The upper byte of each program word is filled with 0x0 or another value specified with the `--fill-upper` option. None of the p-variant assembler directives (including `.pbyte` and `.pword`) are permitted in read-only data sections.

The following examples illustrate how bytes in read-only sections may be accessed:

```
; example 1
mov    #psvpage(L1),w0
mov    w0,PSVPAG          ; set page register
mov    #psvoffset(L1),w0
mov    #psvoffset(L2),w1
mov.b  [w0],w2            ; load the byte at L1
mov.b  [w1],w3            ; load the byte at L2
; example 2
mov    #edspage(L1),w0
mov    w0,DSRPAG          ; set page register
mov    #edsoffset(L1),w0
mov    #edsoffset(L2),w1
mov.b  [w0],w2            ; load the byte at L1
mov.b  [w1],w3            ; load the byte at L2
```

User-defined read-only sections do not require a custom linker script. Based on the `psv` section attribute, the linker will locate the section in program memory and map its labels into the PSV or EDS window. If the programmer wishes to declare a read-only section in a custom linker script, the following syntax may be used:

```
/*
** User-Defined Constants in Program Memory
**
** This section is identified as a read-only section
** by use of the psv section attribute. It will be
** loaded into program memory and mapped into data
** memory using the PSV or EDS window.
*/
userconstants ADDR : AT (LOADADDR)
{
*(userconstants);
} >program
```

In this example, `LOADADDR` specifies the load address in program memory.

It is not generally recommended to define read-only data sections in the linker script. This is because sections that appear in the linker script are allocated sequentially, and read-only data sections have significant page alignment restrictions. Because of these alignment restrictions, sequential allocation can fragment memory and result in less efficient memory utilization.

Likewise, it is not recommended to specify an absolute address for read-only data sections using attributes in source code. Absolute sections also fragment memory and can result in less efficient memory utilization.

11.8 Stack Allocation

The 16-bit device dedicates register W15 for use as a software stack pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. Upon Power-on or Reset, register W15 is initialized to point to a region of memory reserved for the stack. The stack grows upward, towards higher memory addresses.

The 16-bit device also supports stack overflow detection. If the stack limit register SPLIM is initialized, the device will test for overflow on all stack operations. If an overflow should occur, the processor will initiate a stack error exception. By default, this will result in a processor Reset. Applications may also install a stack error exception handler by defining an interrupt function named __StackError. See 11.10. Interrupt Vector Tables [DD] for details.

By default, 16-bit linker allocates the largest stack possible from unused data memory. Therefore, care should be taken when assigning symbols to data sections so as to leave room for the stack (see 5.1. Directives that Define Sections).

The location and size of the stack is reported in the link map output file, under the heading Dynamic Memory Usage. Applications can ensure that at least a minimum sized stack is available by using the `--stack` command option. For example:

```
xc16-ld -o t.exe t1.o --stack=0x100
```

While performing automatic stack allocation, 16-bit linker increases the minimum required size by a small amount to accommodate the processing of stack overflow exceptions. The stack limit register SPLIM is initialized to point just below this extra space, which acts as a stack overflow guardband. If not enough memory is available for the minimum size stack plus guardband, the linker will report an error.

The default stack guardband size is 16 bytes. Applications can specify a different size by using the `--stackguard` command option. For example:

```
xc16-ld -o t.exe t1.o --stackguard=32
```

As an alternative to automatic stack allocation, the stack may be allocated directly with a user-defined section in assembly language. For example:

```
.section my_stack, stack
.space 0x100
```

When the stack is allocated in this way, the usable stack space will be slightly less than 0x100 bytes, since a portion of the user-defined section will be reserved for the stack guardband.

An appropriate worst case stack usage requirement can be determined using the Stack Usage Guidance analysis tool, which can provide a compile-time analysis static of the end executable. For more on this tool, see the "*MPLAB XC16 C Compiler User's Guide*" (DS50002071).

Regardless of how the stack is allocated (automatically or by user-defined section) the linker creates two symbols for use by the startup module. `__SP_init` defines the initial value for the stack pointer (W15), and `__SPLIM_init` defines the initial value for the stack limit register (SPLIM).

The start-up module uses these symbols to initialize the stack pointer and stack pointer limit register. Normally the start-up module is provided by `libpic30.a`. In special cases, the application may provide its own start-up code. The following stack initialization sequence may be used:

```
mov      #__SP_init,w15    ; initialize w15
mov      #__SPLIM_init,w0  ;
mov      w0,_SPLIM         ; initialize SPLIM
```

11.9 Heap Allocation

The 16-bit compiler standard C library, `libc.a`, supports dynamic memory allocation functions such as `malloc()` and `free()`. Applications which utilize these functions must instruct the linker to reserve a portion of 16-bit data memory for this purpose. The reserved memory is called a heap.

Applications can specify the heap size by using the `--heap` command option. For example:

```
xc16-ld -o t.exe t1.o --heap=0x100
```

While performing automatic heap allocation, the linker allocates the heap from unused data memory. The heap size is always specified by the programmer. In contrast, the linker sets the stack size to a maximum value, utilizing all remaining data memory.

As an alternative to automatic heap allocation, the heap may be allocated directly with a user-defined section in assembly source code. For example:

```
.section my_heap, heap
.space 0x100
```

The location and size of the heap are reported in the link map output file, under the heading Dynamic Memory Usage. If the requested size is not available, the linker reports an error.

11.10 Interrupt Vector Tables [DD]

dsPIC30F/33F DSC and PIC24F/H MCU devices have two interrupt vector tables - a primary and an alternate table, each containing exception vectors, as well as a RESET instruction at location zero. By convention, the linker initializes the RESET instruction and interrupt vector tables automatically, using information provided in the standard linker scripts.

The 16-bit compiler provides a special syntax for writing interrupt handlers. See the "*MPLAB® XC16 C Compiler User's Guide*" (DS50002071) for more information.

Assembly language programmers can install interrupt handlers simply by following the standard naming conventions. Interrupt handlers declared with the standard names and defined as globals are automatically installed into the vector tables.

By convention, the entry point named `__reset` takes control at device Reset. All applications written in assembly language must include a Reset function with this name. For C programs, the Reset function is provided in libpic30, which initializes the C run-time environment.

Note: Applications may provide a default interrupt handler, which will be installed into any unused vector table entries. In assembly language, the name of the default interrupt handler is `__DefaultInterrupt`. In C the name is `_DefaultInterrupt`.
If the application does not provide a default interrupt handler, the linker will create one in section `.isr` that contains a reset instruction. Creation of a default interrupt handler by the linker may be suppressed with the `--no-isr` option. In that case unused slots in the interrupt vector tables will be filled with zeros.

11.10.1 Interrupt Handler Example

The following example provides a Reset function and a default interrupt handler in assembly language. The default interrupt handler uses persistent data storage to keep a count of unexpected interrupts and/or error traps.

```
.include "p30f6014.inc"
.text
.global __reset

__reset:
 ;; takes control at device reset/power-on
 mov   #__SP_init,w15      ; initialize stack pointer
 mov   #__SPLIM_init,w0    ;  and stack limit register
 mov   w0,SPLIM          ;
 btst  RCON,#POR         ; was this a power-on reset?
 bra   z,start           ;  branch if not
 clr   FaultCount        ; else clear fault counter
 bclr  RCON,#POR         ;  and power-on bit

start:
 goto  main              ; start application
 .global __T1Interrupt

__T1Interrupt:
 ;; services timer 1 interrupts
 bclr  IFS0,#T1IF        ; clear the interrupt flag
 retfie                  ;  and return from interrupt
 .global __DefaultInterrupt

__DefaultInterrupt:
 ;; services all other interrupts & traps
 inc  FaultCount         ; increment the fault counter
 reset                   ;  and reset the device
 .section .pbss,persist  ; persistent data storage
 .global FaultCount      ;  is not affected by reset

FaultCount:
 .space 2                ; count of unexpected interrupts
```

The standard naming conventions for interrupt handlers are described in the sections below.

Note: The compiler requires only one leading underscore before any of the interrupt handler names. The assembler requires two leading underscores before any of the interrupt handler names. The compiler format is shown in tables in the following sections.

11.10.2 Interrupt Tables Location

For tables of interrupt vectors by device family:

- In MPLAB X IDE, for newer versions of the compiler, open the Dashboard window and click on the Compiler Help button.
- On the command-line, see the `docs` subdirectory of the MPLAB XC16 C compiler install directory (e.g., `C:\Program Files\Microchip\xc16\v1.70`). Open the `XC16MasterIndex` file and click on the “Interrupt Vector Tables Reference” link.

11.11 Optimizing Memory Usage

For memory intensive applications, it is often necessary to optimize memory usage by reducing or eliminating any unused gaps. The linker will optimize memory allocation automatically in most cases. However, certain constructs in source code and/or linker scripts may introduce gaps and should be avoided.

Memory gaps generally fall into the following categories.

11.11.1 Gaps Between Variables of Different Types

Gaps may be inserted between variables of different types to satisfy address alignment requirements. For example, the following sequence of C statements will result in a gap:

```
char c1;
int i;
char c2;
int j;
```

Because the processor requires integers to be aligned on a 16-bit boundary, a padding byte was inserted after variables `c1` and `c2`. To eliminate this padding, variables of the same type should be defined together, as shown:

```
char c1,c2;
int i,j;
```

Gaps between variables are not visible to the linker, and are not reported in the link map. To detect these gaps, an assembly listing file must be created. The following
procedure can be used:

- If the source file is written in C, specify the `-save-temps` command line option to the compiler. This will cause an assembly version of the source file to be saved in *`filename`*`.s`.
  `xc16-gcc test.c -save-temps`
- Specify the `-ai` listing option to the assembler. This will cause a table of section information to be generated.

`xc16-as test.s -ai`

```
SECTION INFORMATION:
Section        Length (PC units)   Length (bytes) (dec)
-------        -----------------   -------------------
.text                         0                0  (0)
TOTAL PROGRAM MEMORY USED (bytes):              0  (0)
Section           Alignment Gaps   Length (bytes) (dec)
-------           --------------   -------------------
.data                         0                0  (0)
.bss                          0                0  (0)
.nbss                       0x2              0x8  (8)
TOTAL DATA MEMORY USED (bytes):           0x8  (8)
```

In this example, 2 bytes of unused memory were inserted into section `.nbss`. Gaps between ordinary C variables will not exceed 1 byte per variable.

11.11.2 Gaps Between Aligned Variables

Variables may be defined in C with the `aligned` attribute in order to specify special alignment requirements for modulo addressing or other purposes. Use of the `aligned` attribute will cause the variable to be allocated in a unique section. Since a unique section is never combined with other input sections, no alignment padding is necessary and the linker will allocate memory for the aligned variable in the most efficient way possible.

For example, the following sequence of C statements will not result in an alignment gap, because variable `buf` is allocated in a unique section automatically:

```
char c1,c2;
int i,j;
int __attribute__((aligned(256))) buf[128];
```

When allocating space for aligned variables in assembly language, the source code must also specify a section name. Unless the aligned variable is defined in a unique section, alignment padding may be inserted. For example, the following sequence of assembly statements would result in a large alignment gap, and should be avoided:

```
.section my_vars,bss
.global _var1,_var2,_buf
_var1: .space 2
_var2: .space 2
; location counter is now 4
```

```
.align 256
_buf:  .space 256
; location counter is now 512
```

Re-ordering the statements so that `_buf` is defined first will not eliminate the gap. A named input section will be padded so that its length is a multiple of the requested alignment. This is necessary in order to guarantee correct alignment when multiple input sections with the same name are combined by the linker. Therefore reordering statements would cause the gap to move, but would not eliminate the gap.

Aligned variables in assembly must be defined in a unique section in order to avoid alignment padding. It is not sufficient to specify a section name that is used only once, because the assembler does not know if that section will be combined with others by the linker. Instead, the special section name `*` should be used. As explained in 5.1. Directives that Define Sections, the section name `*` instructs the assembler to create a unique section that will not be combined with other sections.

To avoid alignment gaps, the previous example could be written as:

```
.section my_vars,bss
.global _var1,_var2
_var1: .space 2
_var2: .space 2
.section *,bss
.global _buf
.align 256
_buf:  .space 256
```

The alignment requirement for `_buf` could also be specified in the `.section` directive, as shown:

```
.section *,bss,align(256)
.global _buf
_buf:  .space 256
```

11.11.3 Gaps Between Input Sections

Gaps between input sections are similar to gaps between aligned variables, except that the padding is inserted by the linker, not the assembler. This type of gap can occur when variables with different alignment requirements are defined in separate source files.

A necessary condition for the insertion of alignment gaps by the linker is explicit mapping of input sections in the linker script. For example, older versions of the 16-bit compiler (prior to version 1.30) included the following definition:

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

This example maps all input sections named `.data` and all input sections named `.dconst` into a single output section. The various input sections will be combined sequentially. If the alignment requirement of any section exceeds that of the previous section, the linker will insert padding as needed and report an alignment gap in the link map:

```
Data Memory Usage
section        address      alignment gaps    total length  (dec)
-------        -------      --------------    --------------------
.data            0x800                0x10            0x90  (144)
Total data memory used (bytes):            0x90  (144) <1%
```

The remedy for this type of gap is to simply eliminate the mapping of input sections in linker scripts. Unmapped sections are allocated individually by the linker, so that no special alignment padding is necessary. Newer versions of the 16-bit compiler (version 1.30 and later) do not explicitly map any input sections in data memory for this reason.

11.11.4 Gaps Between Output Sections

Gaps between output sections can occur when the alignment requirements differ and multiple sections are allocated sequentially into the same memory region.

A necessary condition for the insertion of alignment gaps between output sections is explicit mapping of output sections in the linker script. For example, older versions of the 16-bit compiler (prior to version 1.30) included the following definitions:

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

This example creates two output sections (`.pbss` and `.bss`) and maps them into memory region `data`. Because the output sections are allocated sequentially, any difference in alignment requirements will result in gap.

In some instances the linker will make use of this gap, depending on the availability, size, and alignment requirements of any unmapped sections. In general it is preferable to eliminate the explicit mapping of output sections in linker scripts. When all output sections are unmapped, the linker is free to perform a best-fit allocation based on section attributes.

One consequence of best-fit allocation is that gaps between output sections may appear in unexpected places. The linker tries to use small memory blocks first, and will locate sections to leave the largest unused portions. When memory is segmented, such as by the introduction of an absolute section, the arrangement in memory may change dramatically. This should not be a problem unless the programmer expects a certain area of memory to remain unused. In such cases the programmer should reserve memory explicitly, using an array definition in source code, or by editing the linker script.

Explicit mapping of output sections in linker scripts is recommended only when the proximity or relative ordering of sections is important, and can't be satisfied using the section attributes described in 5.1. Directives that Define Sections.

11.12 Boot and Secure Segments

The linker supports boot, secure, and general segments as described in the "*CodeGuard™ Security Reference Manual*" (DS70000180). The security model which includes segment sizes and configuration options may be specified in multiple ways. The linker allocates memory according to this security model and supports independent linking of application segments.

11.12.1 Specifying the Security Model

The application security model (including the sizes of various secure segments in FLASH, RAM, and EEDATA) can be specified in two ways:

1. In source code using macros currently defined for the FBS, FSS, FGS configuration words. See processor-specific include files for details and examples.
2. Using linker command options (see 9.6. Options that Specify CodeGuard Security Features).

   If both methods are used to provide conflicting information, the linker will issue a diagnostic. Likewise, a diagnostic will be issued if a security model is specified that can not be supported by the target device. The security model will be encoded by the linker into the executable file as contents for the FBS, FSS, and FGS configuration words.

A summary of CodeGuard Security options and segment sizes is written to the link map file. For example:

```
Selected CodeGuard Options:
FBS:BSS:STRD_SMALL_BOOT_CODE
FSS:SSS:STRD_SMALL_SEC_CODE
CodeGuard FLASH Memory:
boot    0x100 to 0x3fe
secure  0x400 to 0x1ffe
general 0x2000 to 0x17ffe
CodeGuard RAM Memory:
general 0x800 to 0x279f
secure  (none)
boot    (none)
```

11.12.2 User-Defined Boot and Secure Segments

User-defined `boot` and `secure` segments are supported in program memory and data memory. This allows an application to take advantage of the CodeGuard Security language extensions on any device, not just CodeGuard Security-enabled devices. User-defined segments are specified with the `ram_size` and `flash_size` options (see 9.6. Options that Specify CodeGuard Security Features).

A summary of user-defined boot and secure segments is written to the link map file. For example:

```
User-Defined CodeGuard Segments
boot RAM:          0x20 bytes
secure RAM:        0x80 bytes
CodeGuard FLASH Memory:
boot    (none)
secure  (none)
general 0x100 to 0x17ffe
CodeGuard RAM Memory:
general 0x800 to 0x26ff
secure  0x2700 to 0x277f
boot    0x2780 to 0x279f
```

User-defined segment options should not be combined with CodeGuard Security options. They are intended for debugging and/or special bootloader applications. User-defined segment options are not encoded in the FBS, FSS, FGS configuration words.

11.12.3 Boot and Secure Segment Allocation

The linker will collect input sections designated as boot or secure and allocate them according to the security model. Diagnostics will be issued for errors such as overflow of a secure segment, or requests for a type of protected memory that does not match the security model.

The linker reserves memory for boot and secure segments by adjusting boundaries of the following memory regions: program, data, and eedata. Therefore the name, origin, and length of these regions expressed in the linker script should reflect the original values, not values adjusted for boot and secure segments.

Note: Only sections explicitly designated as `boot` or `secure` will be allocated in the `boot` and `secure` segments. For independently linked applications, `boot` and `secure` functions must not call any library functions, or have any section dependencies that are not explicitly designated as `boot` or `secure`.

If access entry points have been defined, the linker will construct branch tables as needed for the boot or secure segment. Branch tables fill the entire access area (32 instruction words), regardless of how many access entry slots are actually used. This ensures that secure segment object code can be reached only by access entry point. Unused slots in the branch table will be filled with the default entry if one has been specified.

Execution flow may reach access entry points in several different ways, using a combination of machine instructions and data directives. Each access entry consists of a single, unconditional branch instruction, which targets the actual object code for a secure function.

11.12.4 Resolving Symbols

Symbol references within CodeGuard Security segments, and between CodeGuard Security segments, will be processed normally. If access entry points have been specified in a code address reference or in a function call reference, they will be resolved to specific offsets in the access entry tables. This mechanism allows the linker to

resolve references to `boot` or `secure` functions that are defined only in terms of their access entry slot number, and is the key to supporting independently-linked applications.

Interrupt service routines designated for the `boot` or `secure` segments will be installed as a vector in slot 16 of the appropriate segment. Unused slots in the access entry tables are resolved to the `unused` function handler if one has been defined.

Note: The linker implements the security model in terms of memory allocation, but does not enforce a security policy. For example, references to a function defined in a secure segment from a lower privileged segment are permitted. Therefore it is possible to successfully link an application that fails at runtime due to CodeGuard Security hardware protection. This should be a relatively uncommon occurrence, since in practice strict CodeGuard Security protection implies independently-linked application segments.

11.12.5 Example of Simple Bootloader Application

A simple bootloader might look like this:

```
#include <xc.h>
#include "bootloader.h"
volatile int safe_to_continue = 0;

void __attribute__((interrupt)) _T1Interrupt(void) {
 safe_to_continue = 1;
 IFS0bits.T1IF = 0;
}

/*
* will be filled in by target application
*/
void (*volatile startup_location)(void)  __attribute__((section("startup"),shared));
main() {
 IPC0bits.T1IP = 6;
 IEC0bits.T1IE=1;
 PR1 = 1000;
 T1CONbits.TON = 1;

 while (safe_to_continue == 0);
  T1CONbits.TON = 0;

 // presumably we have no communications request, and can proceed
 if (startup_location) startup_location();
}
```

The variable `startup_location` is shared and defined in the bootloader. The end application may redefine the contents of the variable, but not the address. The target application will provide an over-ridden definition of this value to be the address of its startup location. For a C application, this will be the function `_reset`.

```
#include <xc.h>
#include <stdio.h>
#include "bootloader.h"
extern void _reset(void);
void (*volatile startup_location)(void)  __attribute__((section("startup"),shared)) = &_reset;
main() {
 fprintf(stderr,"Hello world\n");
}
```

The rest of the bootloader application could be filled-in to accept some communication and re-program the target. When the whole system starts, the NULL initialization of `startup_location` may be over-written by the target application.

11.13 Co-resident Application Linking

Co-resident applications are programs that share the same physical memory space on an MCU or DSC. These applications are linked together in such a way that they can share the device memory resource.

11.13.1 Associated Options

There are several optional controls:

- There is only one reset vector and one set of fuses. Only one application can control these locations. The linker command line option `--coresident` asks the linker not to fill these locations, and once they are filled in it would be a link error to try and define them again. See 9.2.5. --coresident
- There is still a restriction that there be only one constant section for the const-in-code memory model, which must be shared amongst all linked applications. It is possible to reserve some space for future growth by using the option `--reserve-const=size`. This option will set aside up to size total bytes of space to be used for the `.const` section in a future link. Remember the total size of the `.const` section cannot be larger than 32K bytes. This option must be used if const memory space is being used in a co-resident application domain. See 9.2.34. --reserve-const=size.
- Padding general FLASH areas. There are already methods to add padding to prelinked, input sections. However, this could be wasteful if all we need to do is ensure that the contiguous allocation ends on a particular boundary. The option `--pad-flash=`*`size`* will ensure that the linked output section is padded to a size byte boundary. See 9.2.29. --pad-flash=size.
- The linker option `--no-isr` has been extended to prevent the linker from completing the vector table. With this option specified, the linked executable will contain entries only for vector slots that are used in the application. The others will remain unfilled and can be defined in a future link. It is safer, in a final application, to have all vector slots completely defined. Without the option `--no-isr`, the linker will fill unused slots to call the `_DefaultInterrupt(void)` handler function, which it will define if none exists. See 9.2.17. --no-isr.
- The linker option `--no-isr` instructs the linker not to generate an IVT or AIVT, unless one is explicitly created in the linker script or by other means. See 9.2.19. --no-ivt.
- The command line option `--application-id=`*`name`* causes the linker to create alias names for each external symbol. The provided name is prepended to the normal symbol name; name should be C appropriate. See 9.2.2. --application-id=name.
- The command line option `--memory-usage` causes the linker to create information about the static memory usage for the linked application space. This information is represented in the executable as a null terminated sequence of pairs of start and end addresses for Flash usage and RAM usage. Flash usage information is placed in a section named `.flash_usage`, RAM usage is in `.ram_usage`. See 9.4.4. --memory-usage and 11.14. Linker Resolved Symbols.

11.13.2 Associated Attributes

An attribute, shared, can be applied to a function or data in C or a section in assembly to indicate that the entry may be used outside of the application. A data item will be initialized at startup of any application in the co-resident set.

11.13.3 Co-resident Usage Restrictions

When using co-resident applications:

- A function may be shared between co-resident applications, but calling such a function should only be attempted if you are sure that all data is initialized, i.e., the called function should only reference shared or constant data.
- Only one co-resident application can fill in each vector slot.
- Each co-resident application will share the same const-in-code Flash PSV page.
- A co-resident application should have a stack provided. For a non-coresident application, the language tool will select the largest block of free space to be the stack. In a co-resident application, this should be defined by the programmer. Fortunately this is easily done with a simple assembly file. The following will reserve 1024 bytes for the stack:

```
.section *,stack
.space 1024
```

11.14 Linker Resolved Symbols

The following symbols are defined by the linker and may be useful in code development.

11.14.1 __DATA_LENGTH , __CODE_LENGTH

Description:

Symbols that represent the maximum length of their respective data sections.

Include:

`libpic30.h`

Prototype:

```
extern int __DATA_LENGTH;

extern int __CODE_LENGTH;
```

Remarks:

These symbols are defined in the default linker scripts. They are treated like assembler equates but can be used from C.

Default Behavior:

The address of the symbol (its value in equate terms) represents the maximum length of the data section.

11.14.2 _PROGRAM_END

Description:

A symbol defined in program memory to mark the highest address used by a CODE or PSV section.

Include:

`libpic30.h`

Prototype:

```
__attribute__((space(prog))) int _PROGRAM_END
```

Remarks:

In C, the symbol should be referenced with the address operator (`&`), as in a built-in function call that accepts the address of an object in program memory. Also, this symbol can be used by applications as an end point for checksum calculations.

In assembly language, it should be referenced with an extra underbar character in the prefix.

Default Behavior:

The highest address used by a CODE or PSV section.

Examples:

C code:

```
__builtin_tblpage(&_PROGRAM_END)
__builtin_tbloffset(&_PROGRAM_END)
_prog_addressT big_addr;
_init_prog_address(big_addr, &_PROGRAM_END)
```

Assembly code:

```
mov #tblpage(__PROGRAM_END),w0
mov #tbloffset(__PROGRAM_END),w1
.pword __PROGRAM_END
.long __PROGRAM_END
```

11.14.3 __TARGET_DIVIDE_CYCLES

Description:

`__TARGET_DIVIDE_CYCLES` resolves to the value 5 or 17 based on the processor Instruction Set Architecture. Using this symbol in assembly code instead of hard-coding a number helps in porting of code from one device to another.

Include:

`libpic30.h`

Prototype:

```
repeat #__TARGET_DIVIDE_CYCLES;
```

Remarks:

The *16-Bit MCU and DSC Programmer's Reference Manual* (DS70000157), DIV.S instruction behavior section,explains how many cycles the div instruction takes, including the repeat, in its instruction description. Under Cycles:

- 18 (plus 1 for REPEAT execution) for PIC24F, PIC24H, PIC24E, dsPIC30F, dsPIC33F,dsPIC33E
- 6 (plus 1 for REPEAT execution) for dsPIC33C

Default Behavior:

The literal value of 5 or 7 is assigned based on device instruction set architecture.

12. Linker Examples

The 16-bit devices include many architectural features that require special handling by the linker. The 16-bit compiler and assembler each provide a syntax than can be used to designate certain elements of an application for special handling. In C, a rich set of attributes are available to modify variable and function definitions (see the "*MPLAB XC16 C Compiler User's Guide*" - DS50002071). In assembly language, variables and functions are abstracted into memory sections, which become inputs to the linker. The assembler provides another set of attributes that are available to modify section definitions (see Section 4.7 "Directives that Modify Section Alignment"assembler documentation concerning directives that modify section alignment).

This chapter includes a number of 16-bit specific linker examples and shows the equivalent syntax in C and assembly language.

12.1 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 data memory and program memory can be determined from the link map output, as shown in this excerpt:

```
...
External Symbols in Data Memory (by address):
                 0x0802 __curbrk
                 0x0804 _Stdin
                 0x082c _Stdout
                 0x0854 _Stderr
                 0x087c _Files
                 0x088c _Aldata
                 0x0890 _Size_block
...
External Symbols in Data Memory (by name):
                 0x0802 __curbrk
                 0x088c _Aldata
                 0x087c _Files
                 0x0890 _Size_block
                 0x0854 _Stderr
                 0x0804 _Stdin
                 0x082c _Stdout
...
```

In some cases it is necessary for the programmer to specify the address where a certain variable or function should be located. Traditionally this is done by creating a user-defined section and writing a custom linker script. The 16-bit assembler and compiler provide a set of attributes that can be used to specify absolute addresses and memory spaces directly in source code. When these attributes are used, custom linker scripts are not required.

Note: By specifying an absolute address, the programmer assumes the 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.

12.2 Locating a Variable at a Specific Address

In this example, array `buf1` is located at a specific address in data memory. The address of `buf1` can be confirmed by executing the program in the simulator, or by examining the link map.

```
#include "stdio.h"
int __attribute__((address(0x900))) buf1[128];
void main()
{
  printf("0x900 = 0x%x\n", &buf1);
}
```

The equivalent array definition in assembly language appears below. The `.align` directive is optional and represents the default alignment in data memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
       .section *,address(0x900),bss,near
       .global _buf1
       .align 2
_buf1: .space 256
```

12.3 Locating a Function at a Specific Address

In this example, function `func` is located at a specific address. Two built-in compiler functions are used to calculate the program memory address, which is not otherwise available in C.

```
#include "stdio.h"
void __attribute__((address(0x2000))) func()
{}
void main()
{
  long addr;
  addr = ((long) __builtin_tblpage(func) << 16)
         + __builtin_tbloffset(func);
  printf("0x2000 = 0x%lx\n", addr);
}
```

The equivalent function definition in assembly language appears below. The `.align` directive is optional and represents the default alignment in program memory. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
       .section *,address(0x2000),code
       .global _func
       .align 2
_func: return
```

12.4 Using More than 32K of Constants

By default, the compiler collects const-qualified variables and string literals into a compiler managed section named `.const`. This section is allocated in program memory, and is mapped into data memory by means of the Program Space Visibility (PSV) window, or the Extended Data Space (EDS) window. Variables may be explicitly assigned to this section with the `space(auto_psv)` attribute.

Because `.const` is a PSV-type section, it is limited to 32K of total constants. To use more constants, variables may be assigned to other sections with the `space(psv)` attribute. This attribute causes the variable to be allocated in a program memory section that is designated for use with the PSV or EDS window.

For example:

```
const int __attribute__((space(psv))) table1[] =
   { 1, 2, 3, /* and so on */ };
```

`space(psv)` specifies the allocation of the variable, but it does not describe how the variable will be accessed. In order to access variables in `space(psv)`, the PSV or EDS page register must be managed so that the correct range of program memory is visible. Two options for managing the page register are available: compiler-managed access, or user-managed access.

12.4.1 Compiler-Managed Access

With this option, the compiler generates additional instruction as needed to save, set, and restore the PSV or EDS window page register. To specify compiler-managed access, add the `__psv__` access qualifier to the variable definition. For example:

```
__psv__ const int __attribute__((space(psv))) table1[] =
   { 1, 2, 3, /* and so on */ };
```

The `__psv__` access qualifier works with any variable allocated in `space(psv)`. It can be used an any 16-bit device, and directs the compiler to generate code automatically for managing the PSV or EDS window page register.

12.4.2 User-Managed Access

User-managed access means that the programmer must write explicit code to save, set, and restore the PSV or EDS window page register. In certain situations, this could result in faster execution speed.

In the following example, the constant `status_string` is located in the compiler-managed PSV section, while the constant `gamma_factor` is located in a separate PSV section.

Note: To modify this example to run on a device which supports the EDS window, replace references to PSVPAG with DSRPAG.

The compiler will initialize the page register only for the compiler-managed PSV section on startup. To properly access `gamma_factor`, you must manually manage the page register. Namely, save the current page value, set the page register to access `gamma_factor`, and restore the original page value after. To determine the correct page value for a constant stored in program memory, use the `__builtin_psvpage()` helper function.

When the page register has been modified to access `gamma_factor`, be careful not to access constants stored in the compiler-managed PSV section, such as string constants used with `printf()`. Any attempts to access constants stored in the compiler-managed PSV section with an incorrect page value will fail.

Note: On devices with less than 16K instruction words, there is only one page and manual management of the page register is not required.

```
#include "stdio.h"
#include "p30fxxxx.h"
const char __attribute__ ((space(auto_psv))) status_string[2][10] =
{"System OK", "Key Made"};
const int __attribute__ ((space(psv))) gamma_factor[3] = {13, 23, 7};
int main(void)
{
 unsigned psv_shadow;
 unsigned key, seed = 17231;

 /* print the first status string */
 printf ("%s\n", status_string[0]);

 /* save the PSVPAG */
 psv_shadow = PSVPAG;

 /* set the PSVPAG for accessing gamma_factor[] */
 PSVPAG = __builtin_psvpage (gamma_factor);

 /* build the key from gamma_factor */
 key = (seed + gamma_factor[0] + gamma_factor[1]) / gamma_factor[2];

 /* restore the PSVPAG for the compiler-managed PSVPAG */
 PSVPAG = psv_shadow;

 /* print the second status message */
 printf ("%s \n", status_string[1]);
}
```

12.5 Locating a Constant at a Specific Address in Program Memory

In this example, the constant `table` is located at a specific address in program memory. When a constant is specifically placed at an address in program memory, it must be placed in its own PSV section using the

space(psv) attribute. If a device has only one PSV page (16K instruction words or less), the (psv) section and (auto_psv) section will share the same PSV page by default.

Note: It is not possible to place a constant at a specific address in Program Memory using the space(auto_psv) attribute. Only the space(psv) attribute may be used to perform this task.

The __builtin_tbladdress() helper function can be used to find the address of a constant stored in program memory. The __psv__ access qualifier is used to specify compiler-managed access.

```
__psv__ const unsigned __attribute__ ((space(psv),
  address (0x2000))) table[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int main(void)
{
 unsigned sum=0, u;
 long addr;

 /* compute the address of table and print it */
 addr = __builtin_tbladdress(table);

 /* print the address of table */
 printf ("table[] is stored at address 0x%lx\n", addr);

 /* sum the values in table[] */
 for (u=0; u<10; u++) {
      sum += table[u];
 }

 /* print the sum */
 printf ("sum is %d\n", sum);
}
```

The equivalent constant definition for the array table in assembly language appears below. The .align directive is optional and represents the default alignment in program memory. Use of * as a section name causes the assembler to generate a unique name based on the source file name.

```
        .section *,address(0x2000),psv
        .global _table
        .align 2
_table:
        .word 0,1,2,3,4,5,6,7,8,9
```

In order to allocate table in data memory, the space(psv) attribute could be changed to space(data). In this case, the specified address would be a data memory address. In the absence of a space attribute, the keyword const directs the C compiler to allocate the variable in the same space as other compiler constants. Constants are allocated in program memory by default, or in data memory if the constants-in-data memory model is selected.

12.6 Locating and Accessing Data in EEPROM Memory

In this example, two arrays are defined in data EEPROM. Table1 is aligned to a 32-bit address, so it will be eligible for erasing or programming using the row programming algorithm. Table2 is defined with standard alignment, so it must be erased or programmed one word at a time. The macro _EEDATA is used to place a variable in the Data EEPROM section of memory and align the variable to the specified byte boundary. This macro is defined in the processor header files for devices which contain data flash. This example is targeted for the dsPIC30F6014 processor, and includes the processor header file p30f6014.h.

The compiler and linker treat Data EEPROM like any other custom-defined (psv) section. The __psv__ access qualifier is used to instruct the compiler to generate the necessary instructions to manage the PSV or EDS page register automatically.

```
/* load SFR definitions and macros */
#include "p30f6014.h"

/* load standard I/O definitions */
#include "stdio.h"

__psv__ unsigned int _EEDATA(32) Table1[16];
```

```
__psv__ unsigned int _EEDATA(2) Table2[4]= {0x1234, 0x5678, 0x9ABC,0xDEF0};

unsigned int i, temp_data[4];
__psv__ unsigned int *ee_rd_ptr;

int main( void )
{
 /* initialize EEPROM read pointer */
 ee_rd_ptr = &Table2[0];

 /* read integer data from EEPROM */
 temp_data[0] = *ee_rd_ptr++;
 temp_data[1] = *ee_rd_ptr++;
 temp_data[2] = *ee_rd_ptr++;
 temp_data[3] = *ee_rd_ptr;

 /* display it */
 for ( i = 0; i < 4; i++)
   printf(" %x", temp_data[i]);
 printf("\n");
}
```

The equivalent array definitions for `Table1` and `Table2` in assembly language appear below. Use of `*` as a section name causes the assembler to generate a unique name based on the source file name.

```
        .global _Table1
        .section *,eedata
        .align 32
_Table1:
        .space 32
        .global _Table2
        .section *,eedata
        .align 2
_Table2:
        .word 0x1234
        .word 0x5678
        .word 0x9ABC
        .word 0xDEF0
```

12.7 Creating an Incrementing Modulo Buffer in X Memory

An incrementing modulo buffer for use in assembly language can be easily defined in C. In this example, the macro `_XBSS` is used to define an array whose memory alignment is the smallest power of two that is greater than or equal to its size. `_XBSS` is defined in the processor header file, which in this example is `p30f6014.h`.

```
#include "p30f6014.h"
#include "stdio.h"
int _XBSS(128) xbuf[50];
void main()
{
 printf("Should be zero: %x\n", (int) &xbuf % 128);
}
```

The equivalent definition in assembly language appears below. The section alignment could have specified with a separate `.align` directive. By using `*` as a section name, the linker is afforded maximum flexibility to allocate memory.

```
       .global _xbuf
       .section *,xmemory,bss,align(128)
_xbuf: .space 100
```

12.8 Creating a Decrementing Modulo Buffer in Y Memory

A decrementing modulo buffer for use in assembly language can be easily defined in C. In this case, the ending address +1 of the array must be aligned. There is not a suitable predefined macro in the processor header files for

this purpose, so variable attributes are specified directly. The far attribute is recommended because Y memory does not fall within the near space on all devices, and the compiler uses a small-data memory model by default.

```
#include "stdio.h"
int __attribute__((space(ymemory), far, reverse(128))) ybuf[50];
void main()
{
 printf("Should be zero: %x\n",((int) &ybuf + sizeof(ybuf)) % 128);
}
```

Note: Notes:
The `reverse()` attribute can be used with constants stored in program memory only if they are located in a PSV section, not the compiler-managed `auto_psv` section.

The `reverse()` attribute can be used with constants stored in Data EEPROM memory.

The equivalent definition in assembly language appears below. Reverse section alignment can only be specified as an argument to the `.section` directive.

```
       .global _ybuf
       .section *,ymemory,reverse(128)
_ybuf: .space 100
```

12.9 Locating the Stack at a Specific Address

By default, the linker allocates a maximum-size stack using the largest unused block of data memory. In cases where it is necessary for the programmer to specify the location and size of the stack explicitly, the stack may be defined in assembly language, using the `stack` attribute:

```
.section my_stack, stack, address(0x1800) .space 0x100
```

When the stack is allocated in this way, the usable stack space will be slightly less than 0x100 bytes, since a portion of the user-defined section will be reserved for the stack guardband.

12.10 Locating and Reserving Program Memory

In this example, a block of program memory is reserved for a special purpose, such as a bootloader. An arbitrary sized function is allocated in the block, with the remaining space reserved for expansion or other purposes.

The following output section definition is added to a custom linker script:

```
BOOT_START = 0xA200;
BOOT_LEN = 0x400;
my_boot BOOT_START :
{
 *(my_boot);
 . = BOOT_LEN; /* advance dot to the maximum length */
} > program
```

Note the “dot assignment” (.=) that appears inside the section definition after the input sections. Dot is a special variable that represents the location counter, or next fill point, in the current section. It is an offset relative to the start of the section. The statement in effect says “no matter how big the input sections are, make sure the output section is full size.”

The following C function will be allocated in the reserved block:

```
void __attribute__((section("my_boot"))) func1()
{
 /* etc. */
}
```

The equivalent assembly language would be:

```
        .section my_boot,code
        .global _func1
_func1:
        ; and so on..
        return
```

If the bootloader is allocated at the start of program memory, a custom linker script is not be required. Instead, the function could be defined with attribute boot. For example:

```
void __attribute__((boot)) func1()
{
 /* and so on.. */
}
```

The equivalent definition in assembly language:

```
        .section *,code,boot
        .global _func1
_func1:
        ; and so on..
        return
```

In this case, program memory will be automatically reserved by specifying a CodeGuard Security™ boot segment in FBS configuration word settings, or by specifying a user-defined boot segment with linker command option. See Section 10.12 “Boot and Secure Segments” for more information.

13. Linker Map File

The linker has the capability to produce map files. These map files list archive files included, memory usage, external symbols, linker script information and memory maps.

13.1 Generation

To generate a map file whether in MPLAB X IDE or on the command line, you will need to specify an option described in Section 8.5 “Options that Modify the Link Map Output.” By default, a map file is written to a `.map` file.

13.2 Contents

The map files produced by the linker consist of the following items.

Table 13-1. Linker Map File Contents

Item	Description
Tool Name and Command	Path and executable name of the linker, as well as command line options used.
Archive Members	The name of any members from archive files that are included in the link
Memory Usage Report	The starting address and length of all output sections in program memory, data memory and dynamic memory
External Symbol Table	All external symbols in data and program memory
Memory Configuration	All of the memory regions defined for the link
Linker Script and Memory Map	Modules, sections and symbols that are included in the link as specified in the linker script

Example 13-1. Map File

The following is an example of a linker map file for a PIC24FJ MCU project. Note that the `.debug_aranges` register list has been shortened to save space.

```
Microchip Technology Inc, v1.70 (B)

Tool Name:
  c:\program files\microchip\xc16\v1.70\bin\bin\..\bin/elf-ld.exe

Command:
  -p24FJ128GA010 \
  --mdfp=C:/Program Files/Microchip/MPLABX/v5.45.25.3496/packs/Microchip/
PIC24F-GA-GB_DFP/1.5.168/xc16 \
  -o \
  dist/default/production/Example3.X.production.elf \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/.. \
  -LC:/Program Files/Microchip/MPLABX/v5.45.25.3496/packs/Microchip/PIC24F-GA-
GB_DFP/1.5.168/xc16/bin/../support/PIC24F/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../lib \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/PIC24E/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/PIC24F/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/PIC24H/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/dsPIC30F/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/dsPIC33C/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/dsPIC33E/gld \
  -Lc:/program files/microchip/xc16/v1.70/bin/bin/../../support/dsPIC33F/gld \
  build/default/production/example3.o \
  --defsym=__MPLAB_BUILD=1 \
  -Tp24FJ128GA010.gld.00 \
  --stack=16 \
  --check-sections \
  --data-init \
```

```
  --pack-data \
  --handles \
  --isr \
  --no-gc-sections \
  --fill-upper=0 \
  --stackguard=16 \
  --no-force-link \
  --smart-io \
  -Map=dist/default/production/Example3.X.production.map \
  --report-mem \
  --memorysummary \
  dist/default/production/memoryfile.xml \
  -start-group \
  --library=lega-pic30-elf \
  --library=m-elf \
  --library=lega-c-elf \
  -end-group \


Optional library libpPIC24Fxxx.a not found
Archive member included because of file (symbol)

c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\liblega-pic30-
elf.a(crt0_standard.o)
                              (__resetPRI)
c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\liblega-pic30-
elf.a(data_init_standard.o)
                              c:/program files/microchip/
xc16/v1.70/bin/bin/../../lib\liblega-pic30-elf.a(crt0_standard.o)
(__data_init_standard)
c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\liblega-pic30-
elf.a(crt_start_mode_normal.Leo)
                              (__crt_start_mode_normal)


xc16-ld 1.70 (B)

Default Code Model: Small
Default Data Model: Large
Default Scalar Model: Small

"program" Memory  [Origin = 0x200, Length = 0x155fc]

section                    address   length (PC units)   length (bytes) (dec)
-------                    -------   -----------------   --------------------
.text                        0x200               0xe4           0x156  (342)
.text                        0x2e4               0x1a            0x27  (39)
.dinit                       0x2fe                0x8             0xc  (12)

                 Total "program" memory used (bytes):          0x189  (393) <1%


"data" Memory  [Origin = 0x800, Length = 0x2000]

section                    address      alignment gaps    total length  (dec)
-------                    -------      --------------    ------------------
.nbss                        0x800                  0             0x2  (2)

                 Total "data" memory used (bytes):            0x2  (2) <1%


Dynamic Memory Usage

region                     address                      maximum length  (dec)
------                     -------                      --------------------
heap                             0                                   0  (0)
stack                        0x802                             0x1ffe  (8190)

                 Maximum dynamic memory (bytes):         0x1ffe  (8190)

Info: Project is using a large data memory model when small data memory model
is sufficient.


External Symbols in Program Memory (by address):
```

```
                  0x000200                  __resetPRI
                  0x00023c                  __psv_init
                  0x000252                  __data_init
                  0x000252                  __data_init_standard
                  0x0002c8                  __T1Interrupt
                  0x0002de                  __DefaultInterrupt
                  0x0002e4                  _main

External Symbols in Program Memory (by name):

                  0x0002de                  __DefaultInterrupt
                  0x0002c8                  __T1Interrupt
                  0x000252                  __data_init
                  0x000252                  __data_init_standard
                  0x00023c                  __psv_init
                  0x000200                  __resetPRI
                  0x0002e4                  _main

Memory Configuration

Name             Origin             Length             Attributes
data             0x000800           0x002000           a !xr
reset            0x000000           0x000004
ivt              0x000004           0x0000fc
_reserved        0x000100           0x000004
aivt             0x000104           0x0000fc
program          0x000200           0x0155fc           xr
CONFIG2          0x0157fc           0x000002
CONFIG1          0x0157fe           0x000002
*default*        0x000000           0xffffffff

Linker script and memory map

LOAD build/default/production/example3.o
                    0x0001                  __MPLAB_BUILD = 0x1
LOAD pPIC24Fxxx
LOAD c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\libfx-elf.a
                    0x157fc                  __CONFIG2 = 0x157fc
                    0x157fe                  __CONFIG1 = 0x157fe
                    0x0200                  __CODE_BASE = 0x200
                    0x155fc                  __CODE_LENGTH = 0x155fc
                    0x0004                  __IVT_BASE = 0x4
                    0x0104                  __AIVT_BASE = 0x104
                    0x0800                  __DATA_BASE = 0x800
                    0x2000                  __DATA_LENGTH = 0x2000

.reset              0x000000          0x4
                    0x000000          0x2 SHORT 0x200 ABSOLUTE (__reset)
                    0x000001          0x2 SHORT 0x4
                    0x000002          0x2 SHORT 0x0 ((ABSOLUTE (__reset) >>
0x10) & 0x7f)
                    0x000003          0x2 SHORT 0x0

.text               0x000200         0xe4
 *(.init)
 .init              0x000200         0x4c c:/program files/microchip/xc16/
v1.70/bin/bin/../../lib\liblega-pic30-elf.a(crt0_standard.o)
                    0x000200                  __resetPRI
                    0x000200                  __reset
                    0x00023c                  __psv_init
 .init              0x00024c         0x7c c:/program files/microchip/xc16/
v1.70/bin/bin/../../lib\liblega-pic30-elf.a(data_init_standard.o)
                    0x000252                  __data_init_standard
                    0x000252                  __data_init
 *(.user_init)
 *(.handle)
 *(.isr*)
 .isr.text          0x0002c8         0x16 build/default/production/example3.o
                    0x0002c8                  __T1Interrupt
 .isr               0x0002de          0x4 default_isr
                    0x0002de                  __DefaultInterrupt
 *(.libc)
 *(.libm)
 *(.libdsp)
 *(.lib*)
 .libpic30_crt_start_mode
```

```
                  0x0002e2          0x2 c:/program files/microchip/xc16/
v1.70/bin/bin/../../lib\liblega-pic30-elf.a(crt_start_mode_normal.Leo)
                  0x0002e2                  __crt_start_mode
                  0x0002e2                  __crt_start_mode_normal

usercode
 *(usercode)

__CONFIG2
 *(__CONFIG2.sec*)

__CONFIG1
 *(__CONFIG1.sec*)

.comment
 *(.comment)

.debug_info       0x000000        0x672
 *(.debug_info)
 .debug_info      0x000000        0x672 build/default/production/example3.o
 *(.gnu.linkonce.wi.*)

.debug_abbrev     0x000000        0x122
 *(.debug_abbrev)
 .debug_abbrev    0x000000        0x122 build/default/production/example3.o

.debug_line       0x000000        0x127
 *(.debug_line)
 .debug_line      0x000000        0x127 build/default/production/example3.o

.debug_frame      0x000000         0x52
 *(.debug_frame)
 .debug_frame     0x000000         0x52 build/default/production/example3.o

.debug_str        0x000000         0x1b
 *(.debug_str)
 .debug_str       0x000000         0x1b build/default/production/example3.o

.debug_loc
 *(.debug_loc)

.debug_macinfo
 *(.debug_macinfo)

.debug_pubnames
                  0x000000         0x2c
 *(.debug_pubnames)
 .debug_pubnames
                  0x000000         0x2c build/default/production/example3.o

.debug_ranges
 *(.debug_ranges)

.debug_aranges    0x000000         0x18
 *(.debug_aranges)
 .debug_aranges
                  0x000000         0x18 build/default/production/example3.o
                    0x0000                  WREG0 = 0x0
                    0x0000                  _WREG0 = 0x0
                    0x0002                  WREG1 = 0x2
                    0x0002                  _WREG1 = 0x2
                    0x0004                  WREG2 = 0x4
                    0x0004                  _WREG2 = 0x4
                     :                         :
                    0x0220                  _UART1 = 0x220
                    0x0230                  UART2 = 0x230
                    0x0230                  _UART2 = 0x230
START GROUP
LOAD c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\liblega-pic30-
elf.a
LOAD c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\libm-elf.a
LOAD c:/program files/microchip/xc16/v1.70/bin/bin/../../lib\liblega-c-elf.a
END GROUP
OUTPUT(dist/default/production/Example3.X.production.elf elf32-pic30)
LOAD default_isr
LOAD data_init
```

```
.debug_pubtypes
                0x000018         0x76
 .debug_pubtypes
                0x000018         0x76 build/default/production/example3.o

__c30_signature
                0x00008e         0x12
 __c30_signature
                0x00008e          0x6 build/default/production/example3.o
 __c30_signature
                0x000094          0x6 c:/program files/microchip/xc16/
v1.70/bin/bin/../../lib\liblega-pic30-elf.a(crt0_standard.o)
 __c30_signature
                0x00009a          0x6 c:/program files/microchip/xc16/
v1.70/bin/bin/../../lib\liblega-pic30-elf.a(data_init_standard.o)

.nbss             0x0800          0x2
 .nbss            0x0800          0x2 build/default/production/example3.o

.config_JTAGEN    0x0157fe          0x2
 .config_JTAGEN
                0x0157fe          0x2 build/default/production/example3.o

.config_IESO      0x0157fc          0x2
 .config_IESO     0x0157fc          0x2 build/default/production/example3.o

.ivt._T1Interrupt
                0x00001a          0x2
 .ivt._T1Interrupt
                0x00001a          0x2 build/default/production/example3.o

.text             0x0002e4         0x1a
 .text            0x0002e4         0x1a build/default/production/example3.o
                0x0002e4                 _main

.dinit            0x0002fe          0x8
 .dinit           0x0002fe          0x8 data_init
```

14. Linker Errors/Warnings

MPLAB XC16 Object Linker generates errors and warnings. A descriptive list of these outputs is shown here.

14.1 Errors

Symbols
% by zero

Modulo by zero is not computable.

/ by zero

Division by zero is not computable.

A
A heap is required, but has not been specified.

A heap must be specified when using Standard C input/output functions.

Address 0x8 of filename section .reset is not within region reset.

This error indicates a problem with the linker script. Normally section `.reset` is created by the linker script and includes a single GOTO instruction. If a linker script is included in the link as an input file, it will augment the built-in script instead of replacing it. Then section `.reset` will be created twice, resulting in an overflow. To correct this error, specify `--script` or `-T` on the link command before the linker script file name.

Address addr of filename section secname is not within region region.

Section *`secname`* has overflowed the memory region to which it was assigned.

C
Cannot access symbol (name) with file register addressing. Value must be less than 8192.

`name` is not located in near address space. A read or write of name could not be resolved with the small data memory model.

Cannot access symbol (name) at an odd address.

Instructions that operate on word-sized data require operands to be allocated at even addresses.

cannot move location counter backwards (from address1 to address2).

The location counter can be advanced but it cannot be moved backwards. An operation is attempting to move it from address1 backwards to address2.

cannot open linker script file name.

Unable to open the specified linker script file. Check the file name and/or the path.

cannot open name:

Cannot open the input file name. Check for correct spelling, extension or path.

cannot PROVIDE assignment to location counter.

The PROVIDE keyword may not be used to make an assignment to the location counter.

Cannot use relocation type reloc on a symbol (name) that is located in an executable section.

An attempt was made to use a symbol in an executable section as a data address. To reference an executable symbol in a data context, the `psvoffset()` or `tbloffset()` operator is required.

Could not allocate data memory.

The linker could not find a way to allocate all of the sections that have been assigned to region ‘data’.

Could not allocate program memory.

The linker could not find a way to allocate all of the sections that have been assigned to region ‘program’.

Could not allocate eedata memory.

The linker could not find a way to allocate all of the sections that have been assigned to region ‘eedata’.

Could not allocate section ‘name’, because ‘ymemory,near’ is not a valid combination on this device.

The linker could not allocate section name because the combination of section attributes [`ymemory`,`near`] is not valid on the current device.

Could not allocate section secname at address addr.

An address has been specified for secname that conflicts with another section or the limit of memory.

Could not allocate section ‘section name’ it is illegal to use the last word of program memory

Using the last word of program memory is illegal and a link error will be generated if you attempt to place any code there.

D

Data region overlaps PSV window (%d bytes).

The data region address range must be less than the start address for the PSV window. This error occurs when the C compiler’s “constants in code” option is selected and more than 32K of data memory is required for program variables.

--data-init and --no-data-init options can not be used together.

`--data-init` creates a special output section named `.dinit` as a template for the run-time initialization of data, `--no-data-init` does not. Only one option can be used.

__DMA_BASE is needed, but not defined (check linker script?)

__DMA_END is needed, but not defined (check linker script?)

The symbols __DMA_BASE and __DMA_END must be defined in order to allocate variables or sections in dma memory. By convention these symbols are defined in the linker script for a particular device, if that device supports dma memory.

E

EOF in comment.

An end-of-file marker (EOF) was found in a comment.

F

op forward reference of section secname.

The section name being used in the operation has not been defined yet.

G

--gc-sections and -r may not be used together.

Do not use `--gc-sections` option which enables garbage collection of unused input sections with the `-r` option which generates relocatable output.

H

--handles and --no-handles options cannot be used together.

`--handles` supports far code pointers; `--no-handles` does not. Only one option can be used.

I

includes nested too deeply.

include statements should be nested no deeper than 10 levels.

Illegal value for DO instruction offset (-2, -1 or 0).

These values are not permitted.

invalid assignment to location counter.

The operation is not a valid assignment to the location counter.

invalid hex number 'num.'

A hexadecimal number can only use the digits 0-9 and A-F (or a-f). The number is identified as a hex value by using 0x as the prefix.

invalid syntax in flags.

The region attribute flags must be `w`, `x`, `a`, `r`, `i` and/or `l`. ('!' is used to invert the sense of any following attributes.) Any other letters or symbols will produce the invalid syntax error.

M

macros nested too deeply.

Macros should be nested no deeper than 10 levels.

missing argument to -m.

The emulation option (`-m`) requires a name for the emulation linker.

N

Near data space has overflowed by num bytes.

Near data space must fit within the lowest 8K address range. It includes the sections `.nbss` for static or non-initialized variables, and `.ndata` for initialized variables.

no input files.

The 16-bit linker requires at least one object file.

non constant address expression for section secname.

The address for the specified section must be a constant expression.

nonconstant expression for name.

`name` must be a constant expression.

non constant address expression specified. Section will be allocated at the current address in the current region.

If a load address is specified for a section in the linker script using the AT (symbol) expression and "symbol" is not defined, a warning will be generated and the section will be allocated at the current address in the current region.

Not enough contiguous memory for section secname.

The linker attempted to reallocate program memory to prevent a read-only section from crossing a PSV page boundary, but a memory solution could not be found.

Not enough memory for heap (num bytes available).

There was not enough memory free to allocate the heap.

Not enough memory for stack (num bytes available).

There was not enough memory free to allocate the minimum-sized stack.

O

object name was created for the processor which is not instruction set compatible with the target processor.

An object file to be linked was created for a different processor family than the link target, and the instruction sets are not compatible.

Odd values are not permitted for a new location counter.

When a `.org` or `.porg` directive is used in a code section, the new location counter must be even. This error also occurs if an odd value is assigned to the special DOT variable.

P

--pack-data and --no-pack-data options cannot be used together.

`--pack-data` fills the upper byte of each instruction word in the data initialization template with data. `--no-pack-data` does not. Only one option can be used.

PSV section secname exceeds 32 Kbytes (actual size = num).

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

R

region region is full (filename section secname).

The memory region *`region`* is full, but section *`secname`* has been assigned to it.

--relax and -r may not be used together.

The option `--relax` which turns relaxation on may not be used with the `-r` option which generates relocatable output.

relocation truncated to fit: PC RELATIVE BRANCH name.

The relative displacement to function name is greater than 32K instruction words. A function call to name could not be resolved with the small code memory model.

relocation truncated to fit: relocation_type name.

The relocated value of name is too large for its intended use.

S

section .handle must be allocated low in program memory.

A custom linker script has organized memory such that section `.handle` is not located within the first 32K words of program memory.

section secname1 [startaddr1—startaddr2] overlaps section secname2 [startaddr1—startaddr2]\n"),

There is not enough region memory to place both of the specified sections or they have been assigned to addresses that result in an overlap.

-shared not supported.

The option `-shared` is not supported by the 16-bit linker.

Symbol (name) is not located in an executable section.

An attempt was made to call or branch to a symbol in a bss, data or readonly section.

syntax error.

An incorrectly formed expression or other syntax error was encountered in a linker script.

U

undefined symbol '_ _reset' referenced in expression.

The library `-lpic30` is required, or some other input file that contains a start-up function. This error may result from a version or architecture mismatch between the linker and library files.

undefined symbol 'symbol' referenced in expression.

The specified symbol has not been defined.

undefined reference to '_Ctype'.

undefined reference to '_Tolotab'.

undefined reference to '_Touptab'.

These errors indicate a version mismatch between include files and library files, or between library files and precompiled object files. Make sure that all object files to be linked have been compiled with the same version of the 16-bit compiler. If you are using a precompiled object or library file from another vendor, request an update that is compatible with the latest version of the compiler.

undefined reference to 'symbol.'

The specified symbol has not been defined. Either an input file has been omitted, a library file is incomplete, a library file requires a symbol from an earlier library, or a circular reference exists between libraries. Circular references can be resolved with the `--start-group`, `--end-group` options.

unrecognized emulation mode: target
Supported emulations:

The specified target is not an emulation mode supported by the linker. The list of supported emulations follows the error message.

unrecognized -a option 'argument.'

The `-a` option is not supported by 16-bit devices; so it is ignored.

unrecognized -assert option 'option.'

The `-assert` option is not supported by 16-bit devices; so it is ignored.

unrecognized option 'option'.

The specified option is not a recognized linker option. Check the option and its usage information with the `--help` option.

op uses undefined section secname.

The section referred to in the operation is not defined.

X
X data space has overflowed by num bytes.

The address range for X data space must be less than the start of Y data space. The start of Y data space is determined by the processor used.

Y
__YDATA_BASE is needed, but not defined.

By convention, the starting address of Y data memory for a particular device is defined in linker scripts using this name. The linker needed this information to allocate a section with `xmemory` or `ymemory` attribute, but could not find it.

14.2 Warnings

A
Addresses specified for READONLY section name are not valid for PSV window.

The application has specified absolute addresses for a read-only section that are not consistent with the PSV window. If two addresses have been specified, the least-significant 15 bits should be identical. Also, the most significant bit of the virtual address should be set.

C
cannot find entry symbol symbol defaulting to value.

The linker can't find the entry symbol, so it will use the first address in the text section. This message may occur if the `-e` option incorrectly contains an equal sign ('=') in the option (i.e., `-e=0x200`).

common of 'name' overridden by definition
defined here.

The specified variable name has been declared in more than one file with one instance being declared as common. The definition will override the common symbol.

common of 'name' overridden by larger common
larger common is here.

The specified variable name has been declared in more than one file with different values. The smaller value will be overridden with the larger value.

common of 'name' overriding smaller common
smaller common is here.

The specified variable name has been declared in more than one file with different values. The first one encountered was smaller and will be overridden with the larger value.

D
data initialization has been turned off, therefore section secname will not be initialized.

The specified section requires initialization, but data initialization has been turned off; so, the initial data values are discarded. Storage for the data sections will be allocated as usual.

data memory region not specified. Using default upper limit of addr.

The linker has allocated a maximum-size stack. Since the data memory region was not specified, a default upper limit was used.

definition of 'name' overriding common
common is here.

The specified variable name has been declared in more than one file with one instance being declared as common. The definition will override the common symbol.

H
--heap option overrides HEAPSIZE symbol.

The `--heap` option has been specified and the HEAPSIZE symbol has been defined but they have different values so the `--heap` value will be used.

I
initial values were specified for a non-loadable data section (name). These values will be ignored.

By definition, a persistent data section implies data that is not initialized; therefore the values are discarded. Storage for the section will be allocated as usual.

M
multiple common of 'name'
previous common is here.

The specified variable name has been declared in more than one file.

N
no memory region specified for section 'secname'.

Section *`secname`* has been assigned to a default memory region, but other non-default regions are also defined.

O
object name was created for the processor and references register name.

An object file to be linked was created for a different processor family than the link target, and references an SFR that may not be compatible.

P
program memory region not specified. Using default upper limit of addr.

The linker has reallocated program memory to prevent a read-only section from crossing a PSV page boundary. Since the program memory region was not specified, a default upper limit was used.

R
READONLY section secname at addr crosses a PSVPAG boundary.

Address *addr* has been specified for a read-only section, causing it to cross a PSV page boundary. To allow efficient access of constant tables in the PSV window, it is recommended that the section should not cross a PSVPAG boundary.

‘-retain-symbols-file’ overrides ‘-s’ and ‘-S’

If the strip all symbols option (`-s`) or the strip debug symbols option (`-S`) is used with `--retain-symbols-file FILE` only the symbols specified in the file will be kept.

S
--stack option overrides STACKSIZE symbol.

The `--stack` option has been specified and the STACKSIZE symbol has been defined but they have different values so the `--stack` value will be used.

T
target processor ‘name’ does not match linker script.

The link target processor specified on the command line does not match the linker script OUTPUT_ARCH command. The processor name specified on the command line takes precedence.

15. MPLAB XC16 Object Archiver/Librarian

The MPLAB XC16 Object Archiver/Librarian creates, modifies and extracts files from archives. This tool is one of several utilities (`xc16-ar`). An “archive” is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called “members” of the archive).

The original files’ contents, mode (permissions), timestamp, owner and group are preserved in the archive, and can be restored on extraction.

The 16-bit archiver/librarian can maintain archives whose members have names of any length; however, if an `f` modifier is used, the file names will be truncated to 15 characters.

The archiver is considered a binary utility because archives of this sort are most often used as “libraries” holding commonly needed subroutines.

The archiver creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier `s`. Once created, this index is updated in the archive whenever the archiver makes a change to its contents (save for the `q` update operation). An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

You may use `xc16-nm -s` or `xc16-nm --print-armap` to list this index table. If an archive lacks the table, another form of the 16-bit archiver/librarian called `xc16-ranlib` can be used to add only the table.

The 16-bit archiver/librarian is designed to be compatible with two different facilities. You can control its activity using command line options or, if you specify the single command line option `-M`, you can control it with a script supplied via standard input.

15.1 Archiver/Librarian and Other Development Tools

The 16-bit librarian creates an archive file from object files created by the 16-bit assembler. Archive files may then be linked by the 16-bit linker with other relocatable object files to create an executable file. See the “*MPLAB XC16 C Compiler User’s Guide*” (DS50002071) for an overview of the tools process flow.

15.2 Feature Set

Notable features of the librarian include:

- Available for Windows
- Command Line Interface

15.3 Input/Output Files

The 16-bit archiver/librarian generates archive files (`.a`). An archive file is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files.

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

```
xc16-ar -omf=coff [options...]
```

```
xc16-ar -omf=elf [options...]
```

Alternatively, the environment variable XC16_OMF may be used to specify object file format for the 16-bit language tools.

15.4 Syntax

```
xc16-ar [-]P[MOD [RELPOS] [COUNT]] ARCHIVE [MEMBER...]
```

```
xc16-ar -M [ <mri-script ]
```

15.5 Options

When you use the 16-bit archiver/librarian with command line options, the archiver insists on at least two arguments to execute: one key letter specifying the operation (optionally accompanied by other key letters specifying modifiers), and the archive name.

`xc16-ar [-]P[MOD [RELPOS][COUNT]] ARCHIVE [MEMBER...]`

Note: Command line options are case sensitive.

Most operations can also accept further *MEMBER* arguments, specifying archive members. Without specifying members, the entire archive is used.

The 16-bit archiver/librarian allows you to mix the operation code `P` and modifier flags *MOD* in any order, within the first command line argument. If you wish, you may begin the first command line argument with a dash.

The `P` keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them.

Table 15-1. Operation to Execute

Option	Function
`d`	Delete modules from the archive. Specify the names of modules to be deleted as *MEMBER*...; the archive is untouched if you specify no files to delete. If you specify the v modifier, the 16-bit archiver/librarian lists each module as it is deleted.
`m`	Use this operation to move members in an archive. The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member. If no modifiers are used with m, any members you name in the *MEMBER* arguments are moved to the end of the archive; you can use the `a`, `b` or `i` modifiers to move them to a specified place instead.
`p`	Print the specified members of the archive, to the standard output file. If the v modifier is specified, show the member name before copying its contents to standard output. If you specify no *MEMBER* arguments, all the files in the archive are printed.
`q`	Append the files *MEMBER*... into *ARCHIVE*.
`r`	Insert the files *MEMBER*... into *ARCHIVE* (with replacement). If one of the files named in *MEMBER*... does not exist, the archiver displays an error message, and leaves undisturbed any existing members of the archive matching that name. By default, new members are added at the end of the file; but you may use one of the modifiers `a`, `b` or `i` to request placement relative to some existing member. The modifier v used with this operation elicits a line of output for each file inserted, along with one of the letters a or r to indicate whether the file was appended (no old member deleted) or replaced.
`t`	Display a table listing the contents of *ARCHIVE*, or those of the files listed in *MEMBER*..., that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group and size, you can request that by also specifying the v modifier. If you do not specify a *MEMBER*, all files in the archive are listed. For example, if there is more than one file with the same name (`fie`) in an archive (`b.a`), then `xc16-ar t b.a fie` lists only the first instance; to see them all, you must ask for a complete listing in `xc16-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 keyletter to specify variations on an operation's behavior.

Table 15-2. 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 v is always created if it did not exist, when you requested an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.
f	Truncate names in the archive. The 16-bit archiver/librarian will normally permit file names of any length. This will cause it to create archives that are not compatible with the native archiver program on some systems. If this is a concern, the f modifier may be used to truncate file names when putting them in the archive.
i	Insert new files before an existing member of the archive. If you use the modifier i, the name of an existing archive member must be present as the *RELPOS* argument, before the *ARCHIVE* specification. (Same as b.)
l	This modifier is accepted but not used.
N	Uses the *COUNT* parameter. This is used if there are multiple entries in the archive with the same name. Extract or delete instance *COUNT* of the given name from the archive.
o	Preserve the original dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.
P	Use the full path name when matching names in the archive. The 16-bit archiver/librarian cannot create an archive with a full path name (such archives are not *POSIX* compliant), but other archive creators can. This option will cause the archiver to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.
s	Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running xc16-ar s on an archive is equivalent to running xc16-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, xc16-ar r... inserts all files listed into the archive. If you would like to insert only those of the files you list that are newer than existing members of the same names, use this modifier. The u modifier is allowed only for the operation r (replace). In particular, the combination qu is not allowed, since checking the timestamps would lose any speed advantage from the operation q.
v	This modifier requests the verbose version of an operation. Many operations display additional information, such as, file names processed when the modifier v is appended.
V	This modifier shows the version number of the 16-bit archiver/librarian.

15.6 Scripts

If you use the single command line option -M with the archiver, you can control its operation with a rudimentary command language.

```
xc16-ar -M [ <SCRIPT ]
```

Note: Command line options are case sensitive.

This form of the 16-bit archiver/librarian operates interactively if standard input is coming directly from a terminal. During interactive use, the archiver prompts for input (the prompt is AR >), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and the 16-bit archiver/librarian abandons execution (with a nonzero exit code) on any error.

The archiver command language is not designed to be equivalent to the command line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to the 16-bit archiver/librarian for developers who already have scripts written for the MRI “librarian” program.

The syntax for the 16-bit archiver/librarian command language is straightforward:

- commands are recognized in upper or lower case; for example, LIST is the same as list. In the following descriptions, commands are shown in upper case for clarity.
- a single command may appear on each line; it is the first word on the line.
- empty lines are allowed, and have no effect.
- comments are allowed; text after either of the characters “*” or “;” is ignored.
- Whenever you use a list of names as part of the argument to an xc16-ar command, you can separate the individual names with either commas or blanks. Commas are shown in the explanations below, for clarity.
- “+” is used as a line continuation character; if “+” appears at the end of a line, the text on the following line is considered part of the current command.

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

Table 15-3. 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 xc16-ar -d ARCHIVE MODULE ... MODULE. Requires prior use of OPEN or CREATE.

.....continued

Option	Function
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 xc16-ar -t ARCHIVE MODULE.... When verbose output is on, the listing is like xc16-ar -tv ARCHIVE MODULE.... Output normally goes to the standard output stream; however, if you specify OUTPUTFILE as a final argument, the 16-bit archiver/librarian directs the output to that file.
END	Exit from the archiver with a 0 exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last SAVE command, those changes are lost.
EXTRACT MODULE, MODULE, ... MODULE	Extract each named MODULE from the current archive, writing them into the current directory as separate files. Equivalent to xc16-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 xc16-ar tv ARCHIVE. (This single command is a 16-bit archiver/librarian enhancement, rather than present for MRI compatibility.) Requires prior use of OPEN or CREATE.
OPEN ARCHIVE	Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect ARCHIVE until you next use SAVE.
REPLACE MODULE, MODULE, ... MODULE	In the current archive, replace each existing MODULE (named in the REPLACE arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist. Requires prior use of OPEN or CREATE.
VERBOSE	Toggle an internal flag governing the output from DIRECTORY. When the flag is on, DIRECTORY output matches output from xc16-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.

16. Other Utilities

In addition to the archiver/librarian, other utilities are tools available for use with the assembler and/or linker.

Table 16-1. Available Utilities

Utility	Description
`xc16-bin2hex`*	Converts a linked object file into an Intel® hex file.
`xc16-nm`	Lists symbols from an object file.
`xc16-objdump`	Displays information about object files.
`xc16-ranlib`	Generates an index from the contents of an archive and stores it in the archive.
`xc16-strings`	Prints the printable character sequences.
`xc16-strip`	Discards all symbols from an object file.
*See the "*MPLAB XC16 C Compiler User's Guide*" (DS50002071) for an overview of the tools process flow.	

See utility details in the following sections.

16.1 xc16-bin2hex Utility

The binary-to-hexadecimal (`xc16-bin2hex`) utility converts binary files (from the 16-bit linker) to Intel hex format files, suitable for loading into device programmers.

16.1.1 Input/Output Files

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

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

```
xc16-bin2hex -omf=coff file1.out
```

```
xc16-bin2hex -omf=elf file2.out
```

Alternatively, the environment variable `XC16_OMF` may be used to specify object file format for the dsPIC30F language tools.

Because the Intel hex file format is byte-oriented, and the 16-bit PC is not, program memory sections require special treatment. Each 24-bit program word is extended to 32 bits by inserting a so-called "phantom byte". Each program memory address is multiplied by 2 to yield a byte address.

For example, a section that is located at 0x100 in program memory will be represented in the hex file as 0x200. Consider the following assembly language source:

```
; file test.s
.section foo,code,address(0x100)
.pword 0x112233
```

The following commands will assemble the source file and create an Intel hex file:

```
xc16-as -o test.o test.s
xc16-bin2hex test.o
```

The file "test.hex" will be produced, with the following contents:

```
:020000040000fa
:04020000332211096
:00000001FF
```

Notice that the data record (line 2) has a load address of 0200, while the source code specified address 0x100. Note also that the data is represented in “little-endian” format, meaning the least significant byte appears first. The phantom byte appears last, just before the checksum.

16.1.2 Syntax

Command line syntax is:

`xc16-bin2hex` *`object_file`* `[-v] [-a] [-u] [-omf=`*`format`*`]`

Example 16-1. hello.cof

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

```
xc16-bin2hex hello.cof
```

16.1.3 Options

The following options are supported.

Table 16-2. xc16-bin2hex Options

Option	Function
`object_file` `-a`	Sort the contents of the object file in ascending address order. For a summary of the object file contents, add the `-v` option (`-va`).
`-omf=`*`format`*	Specify object file format. The following formats are supported: ELF, COFF. Format names are case-insensitive. ELF in the default.
`-u`	Use upper-case hexadecimal digits
`-v`	Print a table of diagnostic information to standard output in the format shown in the example below.

Example 16-2. -va Option Output

```
writing hello.hex

section  PC address  byte address  length (w/pad)  actual length  (dec)
-------  ----------  ------------  --------------  --------------------
.reset            0             0             0x8            0x6  (6)
.ivt            0x4           0x8            0xf8           0xba  (186)
.aivt          0x84         0x108            0xf8           0xba  (186)
.text         0x100         0x200           0xaec          0x831  (2097)
.const        0x676         0xcec            0x10            0xc  (12)
.dinit        0x67e         0xcfc           0x104           0xc3  (195)
.text         0x700         0xe00            0x14            0xf  (15)
.isr          0x70a         0xe14             0x4            0x3  (3)
Total program memory used (bytes):                         0xa8c  (2700)
```

16.2 xc16-nm Utility

The `xc16-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.

16.2.1 Input/Output Files

- Input: Object archive files
- Output: Object archive files. If no object files are listed as arguments, `xc16-nm` assumes the file `a.out`.

16.2.2 Syntax

Command line syntax is:

```
xc16-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... ]
```

16.2.3 Options

Long and short forms of options, shown in the table below as alternatives, are equivalent.

Table 16-3. xc16-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 `xc16-nm` and exit.
`-l` `--line-numbers`	For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry that refers to the symbol. If line number information can be found, print it after the other symbol information.

`-n`
`-v`
`--numeric-sort`
	Sort symbols numerically by their addresses, rather than alphabetically by their names.

`-omf=`*`format`*	Specify object file format. The following formats are supported: ELF, COFF. Format names are case-insensitive. ELF in the default.
`-p` `--no-sort`	Do not bother to sort the symbols in any order; print them in the order encountered.
`-P` `--portability`	Use the POSIX.2 standard output format instead of the default format. Equivalent to `-f posix`.

.....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 `xc16-ar` or `xc16-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 `xc16-nm` and exit.

16.2.4 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 16-4. 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 BSS).
`C`	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
`D`	The symbol is in the initialized data section.
`N`	The symbol is a debugging symbol.
`R`	The symbol is in a read only data section.
`T`	The symbol is in the text (code) section.
`U`	The symbol is undefined.
`V`	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
`W`	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
`?`	The symbol type is unknown, or object file format specific.

Example 16-3. xc16-nm output

```
00000474 T _fclose
0000023e T _fputc
000001b2 T _fputs
0000051e T _free
```

```
00000700 T _main
000003bc T _malloc
00000334 T _memcpy
00000198 T _puts
0000061a W _remove
0000062c W _sbrk
00000326 T _strlen
00000310 T _strrchr
000005a0 W _write
```

16.3 xc16-objdump Utility

The `xc16-objdump` utility displays information about one or more object files. The options control what particular information to display.

16.3.1 Input/Output Files

- Input: Object archive files
- Output: Object archive files. If no object files are listed as arguments, `xc16-nm` assumes the file `a.out`.

16.3.2 Syntax

Command line syntax is:

```
xc16-objdump [ -a | --archive-headers ]
      [ -d | --disassemble ]
      [ -D | --disassemble-all ]
      [ -EB | -EL | --endian={big | little } ]
      [ -f | --file-headers ]
      [ --file-start-context ]
      [ -g | --debugging ]
      [ -h | --section-headers | --headers ]
      [ -H | --help ]
      [ -j name | --section=name ]
      [ -l | --line-numbers ]
      [ -M options | --disassembler-options=options]
      [-omf=format]
      [ --prefix-addresses]
      [ --psrd-psrd-check]
      [ -r | --reloc ]
      [ -s | --full-contents ]
      [ -S | --source ]
      [ --[no-]show-raw-insn ]
      [ --start-address=address ]
      [ --stop-address=address ]
      [ -t | --syms ]
      [ -V | --version ]
      [ -w | --wide ]
      [ -x | --all-headers ]
      [ -z | --disassemble-zeroes ]
      OBJFILE...
```

`OBJFILE`... are the object files to be examined. When you specify archives, `xc16-objdump` shows information on each of the member object files.

16.3.3 Options

The long and short forms of options, shown in the table below, as alternatives, are equivalent. At least one of the following options `-a`, `-d`, `-D`, `-f`, `-g`, `-G`, `-h`, `-H`, `-p`, `-r`, `-R`, `-S`, `-t`, `-T`, `-V` or `-x` must be given.

Table 16-5. xc16-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 `xc16-ar tv`, `xc16-objdump -a` shows the object file format of each archive member.
`-d` `--disassemble`	Display the assembler mnemonics for the machine instructions from OBJFILE. This option only disassembles those sections that are expected to contain instructions.
`-D` `--disassemble-all`	Like `-d`, but disassemble the contents of all sections, not just those expected to contain instructions.

`-EB`
`-EL`
`--endian={big|little}`
	Specify the endianness of the object files. This only affects disassembly. This can be useful when disassembling a file format that does not describe endianness information, such as S-records.

`-f` `--file-header`	Display summary information from the overall header of each of the OBJFILE files.
`--file-start-context`	Specify that when displaying inter-listed source code/disassembly (assumes '-S') from a file that has not yet been displayed, extend the context to the start of the file.
`-g` `--debugging`	Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C like syntax. Only certain types of debugging information have been implemented.

`-h`
`--section-header`
`--header`
	Display summary information from the section headers of the object file.

`-H` `--help`	Print a summary of the options to `xc16-objdump` and exit.
`-j name` `--section=name`	Display information only for section name.
`-l` `--line-numbers`	Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with `-d`, `-D` or `-r`.
`-M options` `--disassembler-options=`*options*	Pass target specific information to the disassembler. The dsPIC30F device supports the following target specific options: symbolic - Will perform symbolic disassembly.
`-omf=format`	Specify object file format. The following formats are supported: ELF, COFF. Format names are case-insensitive. ELF in the default.
`--prefix-addresses`	When disassembling, print the complete address on each line. This is the older disassembly format.
`--psrd-psrd-check` `[=`*library*`]`	Check for back-to-back data flash reads. Specifying the optional `=library` should be used on unlinked object files (such as a library). This option can be combined with `-d` to get a disassembly listing with additional information.
`-r` `--reloc`	Print the relocation entries of the file. If used with `-d` or `-D`, the relocations are printed interspersed with the disassembly.

.....continued

Option	Function
-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 xc16-nm program.
-V --version	Print the version number of xc16-objdump and exit.
-w --wide	Format some lines for output devices that have more than 80 columns.
-x --all-header	Display all available header information, including the symbol table and relocation entries. Using -x is equivalent to specifying all of -a -f -h -r -t.
-z --disassemble-zeroes	Normally, the disassembly output will skip blocks of zeroes. This option directs the disassembler to disassemble those blocks, just like any other data.

Example 16-4. -h OUTPUT

```
hello.out:     file format coff-pic30

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .reset        00000004  00000000  00000000  00000288  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  1 .text         00000576  00000100  00000100  00000290  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  2 .comment      0000005e  00000000  00000000  00000d7c  2**1
                  CONTENTS, NEVER_LOAD
  3 .ivt          0000007c  00000004  00000004  00000e38  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  4 .aivt         0000007c  00000084  00000084  00000f30  2**1
                  CONTENTS, ALLOC, LOAD, CODE
  5 __c30_signature 0000007e  0000005e  0000005e  00001028  2**1
                  CONTENTS, DEBUGGING
  6 .data         0000008e  00000800  00000800  00001124  2**1
                  CONTENTS, ALLOC, DATA, NEVER_LOAD
  7 .bss          00000002  0000088e  0000088e  00000000  2**1
                  ALLOC
  8 .data         00000002  00000890  00000890  00001240  2**1
                  CONTENTS, ALLOC, DATA, NEVER_LOAD
  9 .bss          00000002  00000892  00000892  00000000  2**1
                  ALLOC
 10 .heap         00000080  00000894  00000894  00000000  2**1
                  ALLOC
 11 .const        00000008  00008676  00000676  00001244  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, PSV
 12 .dinit        00000082  0000067e  0000067e  00001254  2**1
```

```
                  CONTENTS, ALLOC, LOAD, CODE
 13 .text         0000000a  00000700  00000700  00001358  2**1
                  CONTENTS, ALLOC, LOAD, CODE
 14 .isr          00000002  0000070a  0000070a  0000136c  2**1
                  CONTENTS, ALLOC, LOAD, CODE
```

16.4 xc16-ranlib Utility

The `xc16-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 `xc16-nm -s` or `xc16-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 `xc16-ranlib` is completely equivalent to executing `xc16-ar -s` (i.e., the 16-bit archiver/librarian with the `-s` option).

16.4.1 Input/Output Files

- Input: Archive files
- Output: Archive files

16.4.2 Syntax

Command line syntax is:

`xc16-ranlib [-omf=`*`format`*`] [-v | -V | --version] ARCHIVE`

16.4.3 Options

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

Table 16-6. xc16-ranlib Options

Option	Function
`-omf=`*`format`*	Specify object file format. The following formats are supported: ELF, COFF. Format names are case-insensitive. ELF in the default.

`-v`
`-V`
`--version`
	Show the version number of `xc16-ranlib.`

16.5 xc16-strings Utility

For each file given, the `xc16-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.

`xc16-strings` is mainly useful for determining the contents of non-text files.

16.5.1 Input/Output Files

- Input: Any files
- Output: Standard output

16.5.2 Syntax

Command line syntax is:

```
xc16-strings [-a | --all | -] [-f | --print-file-name]
             [--help] [-min-len | -n min-len | --bytes=min-len]
```

```
            [-omf=format] [-t radix | --radix=radix]
            [-v | --version] FILE...
```

16.5.3 Options

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

Table 16-7. xc16-strings Options

Option	Function
-a --all -	Do not scan only the initialized and loaded sections of object files; scan the whole files.
-f --print-file-name	Print the name of the file before each string.
--help	Print a summary of the program usage on the standard output and exit.
-min-len -n *min-len* --bytes=*min-len*	Print sequences of characters that are at least *-min-len* characters long, instead of the default 4.
-omf=*format*	Specify object file format. The following formats are supported: ELF, COFF. Format names are case-insensitive. ELF in the default.
-t *radix* --radix=*radix*	Print the offset within the file before each string. The single character argument specifies the radix of the offset: o for octal, x for hexadecimal, or d for decimal.
-v --version	Print the program version number on the standard output and exit.

16.6 xc16-strip Utility

The xc16-strip utility discards all symbols from the object and archive files specified. At least one file must be given. xc16-strip modifies the files named in its argument, rather than writing modified copies under different names.

16.6.1 Input/Output Files

- Input: Object or archive files
- Output: Object or archive files. If no object or archive files are listed as arguments, xc16-strip assumes the file a.out.

16.6.2 Syntax

Command line syntax is:

```
xc16-strip [ -g | -S | --strip-debug ] [ --help ]
           [ -K symbolname | --keep-symbol=symbolname ]
           [ -N symbolname | --strip-symbol=symbolname ]
           [ -o file ] [-omf=format]
           [ -p | --preserve-dates ]
           [ -R sectionname | --remove-section=sectionname ]
           [ -s | --strip-all ] [--strip-unneeded]
           [ -v | --verbose ] [ -V | --version ]
           [ -x | --discard-all ] [ -X | --discard-locals ]
           OBJFILE...
```

16.6.3 Options

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

Table 16-8. xc16-strip Options

Option	Function

`-g`
`-S`
`--strip-debug`
	Remove debugging symbols only.

`--help`	Show a summary of the options to `xc16-strip` and exit.
`-K` *`symbolname`* `--keep-symbol=`*`symbolname`*	Keep only symbol *`symbolname`* from the source file. This option may be given more than once.
`-N` *`symbolname`* `--strip-symbol=`*`symbolname`*	Remove symbol *`symbolname`* from the source file. This option may be given more than once, and may be combined with strip options other than `-K`.
`-o file`	Put the stripped output in file, rather than replacing the existing file. When this argument is used, only one OBJFILE argument may be specified.
`-omf=`*`format`*	Specify object file format. The following formats are supported: ELF, COFF. Format names are case-insensitive. ELF in the default.
`-p` `--preserve-dates`	Preserve the access and modification dates of the file.
`-R` *`sectionname`* `--remove-section=`*`sectionname`*	Remove any section named *`sectionname`* from the output file. This option may be given more than once. Note that using this option inappropriately may make the output file unusable.
`-s` `--strip-all`	Remove all symbols.
`--strip-unneeded`	Remove all symbols that are not needed for relocation processing.
`-v` `--verbose`	Verbose output: list all object files modified. In the case of archives, `xc16-strip -v` lists all members of the archive.
`-V` `--version`	Show the version number for `xc16-strip`.
`-x` `--discard-all`	Remove non-global symbols.
`-X` `--discard-locals`	Remove compiler-generated local symbols. (These usually start with L or ".".)

17. Deprecated Features

The features described below are considered to be obsolete and have been replaced with more advanced functionality. Projects which depend on deprecated features will work properly with versions of the language tools cited. The use of a deprecated feature will result in a warning; programmers are encouraged to revise their projects in order to eliminate any dependency on deprecated features. Support for these features may be removed entirely in future versions of the language tools.

17.1 Assembler Directives that Define Sections

The following `.section` directive format was deprecated in v1.30. The new directive format may be found in 5.1. Directives that Define Sections.

17.1.1 section name [, “flags”]

Definition

Assembles the following code into a section named name. If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized:

`b`	bss section (uninitialized data)
`n`	Section is not loaded
`d`	Data section (initialized data)
`r`	Read-only data section (PSV window)
`x`	Executable section

If the `n` flag is used by itself, the section defaults to uninitialized data.

If no flags are specified, the default flags depend on the section name. If the section name is not recognized, the default will be for the section to be loadable data.

The following section names are recognized:

Table 17-1. Section Names

Section Name	Default Flag
`.text`	`x`
`.data`	`d`
`.bss`	`b`

Note: Ensure that double quotes are used around flags. If the optional argument to the `.section` directive is not quoted, it is taken as a sub-section number. Remember, a single character in single quotes (i.e., ‘b’) is converted by the preprocessor to a number.

Example 17-1. Section Use

```
.section .const, "r"
 ; The following symbols (C1 and C2) will be placed
 ; in the named section ".const".
C1: .word 0x1234
C2: .word 0x5678
```

17.2 Reserved Section Names with Implied Attributes

Implied attributes for the section names in the table below were deprecated in v1.30.

Reserved Name	Implied Attribute(s)
.xbss	bss, xmemory
.xdata	data, xmemory
.nbss	bss, near
.ndata	data, near
.ndconst	data, near
.pbss	bss, persist
.dconst	data
.ybss	bss, ymemory
.ydata	data, ymemory
.const	psv
.eedata	eedata

See 5.1. Directives that Define Sections for more information.

17.3 Environmental Variables

The environment variable PIC30_OMF was used to specify object file format for the 16-bit language tools. Now use XC16_OMF.

18. GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML

for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

19. Document Revision History

The following is a list of changes by version to this document.

Note: Some revision letters are not used, such as I and O, as they can be confused for numbers in some fonts, and so were skipped.

19.1 Revision G (January 2022)

- Section 10.16 “Notable Symbols” renamed to “Linker Resolved Symbols”. Section 10.16.3 “__TARGET_DIVIDE_CYCLES” added.

19.2 Revision F (February 2021)

- Section 4.2.7.3 “Reserved Section Names with Implied Attributes”: Changed “33” to represent all dsPIC33x devices.
- Section 8.4.23 “`--msecondary-id`, `--msecondary-id-location`” - Added options related to main/secondary core devices.
- Section 10.10 “Stack Allocation” - Added description of Stage Usage Guidance analysis tool.

19.3 Revision E (December 2019)

- Section 8.4.1 “`--add-flags-code=`, `--add-flags-data=`”: Added options.
- Section 8.4.47 “`--wrap symbol`”: Updated `__wrap_symbol` to `_wrap_symbol` and `__real_symbol` to `_real_symbol`. Includes updated example.

19.4 Revision D (February 2018)

- Section 3.8 “Special Operators” - Table 3-8 updated with footnote that two operators cannot be used in an expression.
- Chapter 4. “Assembler Directives” - Numbered all directive sections for better reference.
- Section 4.2.7.2 “Attributes that Modify Section Types” - Table 4-2 updated for ymemory and dma attributes also applicable to dsPIC33EP devices.
- Section 8.2 “Options that Control Output File Creation” - added `--ivt` and `--no-ivt` linker options.
- Section 8.4 “Options that Control Informational Output” - added `--no-psrd-psrd-check` linker option.
- Section 9.5.5.7 “Output Section Data” - removed reference to QUAD; not used.
- Section 10.12.5 “Example of Simple Bootloader Application” - added example.
- Chapter 12. “Linker Map File” - updated linker map file.
- Section 15.3 “xc16-objdump Utility” - added `--psrd-psrd-check[=library]` option.

19.5 Revision C (August 2016)

- Section 4.1 “Directives that Define Sections” - Added to “Attributes that Represent Section Types”: `packedflash`; added to “Attributes that Modify Section Types”: `shared`, `preserved`, `update`, and `priority`.
- Section 8.2 “Options that Control Output File Creation” - Added options `--application-id`, `--coresident`, `--no-ivt`, `--pad-flash`, `--preserve`, `--preserve-all`, `--reserve-const`; Updated option `--no-isr`.
- Section 8.4 “Options that Control Informational Output” - Added option `--memory-usage`.
- Section 10.13 “Co-resident Application Linking” - Co-resident applications information and options.

19.6 Revision B (December 2014)

- Section 4.1 “Directives that Define Sections” - Added notes to .bss and .data to warn against using all data memory for symbols so there is no room for stack.
- Chapter 8. “Linker Command Line Options” - Changed title from “Linker Command Line Interface”. Also placed option lists into tables to better highlight option/no option pairs.
- Section 8.2.23 “`-mreserve`” - Added this section.
- Section 8.2.47 “`--wrap symbol`” - Added more to this section.
- Section 8.3.6 “`--local-stack`” and Section 8.3.7 “`--no-local-stack`” - Added these sections.
- Section 10.8 “Stack Allocation” - Added text to warn against using all data memory for symbols so there is no room for stack.
- Section 13.1 “Errors” - Updated definition for “undefined reference to ‘symbol’”.

19.7 Revision A (September 2013)

Initial release of this document.

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

Microchip Devices Code Protection Feature

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

- 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, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- 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. Microchip is committed to continuously improving the code protection features of our products.

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications 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. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

Trademarks

ISBN: 978-1-5224-9637-3

Quality Management System

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

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
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	**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	**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	**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