MPLAB® XC8 PIC® Assembler 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

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® XC8 PIC® Assembler User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	Select File and then Save.
Quotes	A field name in a window or dialog	“Save project before build”
Underlined, italic text with right angle bracket	A menu path	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
N‘Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4‘b0010, 2‘hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain Courier New	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`C:\Users\User1\Projects`
	Keywords	`static`, `auto`, `extern`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, ‘A’`
Italic Courier New	A variable argument	*`file.o`*, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`xc8 [`*`options`*`] `*`files`*
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	*`var_name`* `[, `*`var_name`*`...]`

	Represents code supplied by user	`void main (void)`
		`{ ...`
		`}`

1.2 Recommended Reading

This user's guide describes the use and features of the MPLAB XC8 PIC Assembler. The following Microchip documents are available and recommended as supplemental reference resources.

MPLAB® XC8 PIC Assembler Migration Guide
This guide is for customers who have MPASM projects and who wish to migrate them to the MPLAB XC8 PIC assembler. It describes the nearest equivalent assembler syntax and directives for MPASM code.

MPLAB® XC8 PIC Assembler Guide For Embedded Engineers
This guide is a getting started guide, describing example projects and commonly used coding sequences used by the MPLAB XC8 PIC assembler. Use this guide if you need to develop new projects using the assembler.

MPLAB® XC8 C Compiler Release Notes for PIC MCU
For the latest information on changes and bug fixes to this assembler, read the Readme file in the docs subdirectory of the MPLAB XC8 installation directory.

Development Tools Release Notes
For the latest information on using other development tools, read the tool-specific Readme files in the docs subdirectory of the MPLAB X IDE installation directory.

2. Assembler Overview

The MPLAB XC8 PIC Assembler is a free-standing cross assembler and linker package, supporting all 8-bit PIC® microcontrollers.

The internal assembler application used by the PIC Assembler is the same as that used by the MPLAB XC8 C Compiler tool, with the assembly language being common between both tools. This document describes the use of the PIC Assembler for writing and building source code written entirely in assembly. If you need assistance with writing assembly code to be linked with C source code, or you are writing assembly code that is to be inserted in-line with C code, refer to *MPLAB® XC8 C Compiler User's Guide for PIC® MCU* (DS50002737) for information that is better suited to these tasks.

2.1 Device Description

This guide describes the MPLAB XC8 PIC Assembler's support for all 8-bit Microchip PIC devices with baseline, enhanced baseline, mid-range, enhanced mid-range and PIC18 cores. Check the `ARCH` field in the device's INI file to confirm the core architecture used by the assembler when building code. The information contained within the INI files typically comes from your selected Device Family Pack (DFP), but the assembler maintains a default set of INI files in the MPLAB XC8 `pic/dat/ini` directory. The following descriptions indicate the distinctions within those device cores.

The baseline core uses a 12-bit-wide instruction set and is available in PIC10, PIC12 and PIC16 part numbers (`ARCH` value of `PIC12`).

The enhanced baseline core also uses a 12-bit instruction set, but this set includes additional instructions. Some of the enhanced baseline chips support interrupts and the additional instructions used by interrupts. These devices are available in PIC12 and PIC16 part numbers (`ARCH` value of `PIC12E`, or `PIC12IE` for those with interrupt support).

The mid-range core uses a 14-bit-wide instruction set that includes more instructions than the baseline core. It has larger data memory banks and program memory pages, as well. It is available in PIC12, PIC14 and PIC16 part numbers (`ARCH` value of `PIC14`).

The Enhanced mid-range core also uses a 14-bit-wide instruction set but incorporates additional instructions and features. There are both PIC12 and PIC16 part numbers that are based on the Enhanced mid-range core (`ARCH` value of `PIC14E` or `PIC14EX`).

The PIC18 core instruction set is 16 bits wide and features additional instructions and an expanded register set. PIC18 core devices have part numbers that begin with PIC18. Some PIC18 devices implement extended data memory and a vectored interrupt controller module with support for one or more interrupt vector tables, rather than fixed-location, dual priority vectors (`ARCH` value of `PIC18`, or `PIC18XV` for those with the extended data memory and the vectored interrupt controller module).

See 3.4.20. Print-devices for information on finding the full list of devices that are supported by the assembler. Support for a new device might be possible after downloading an updated Device Family Pack.

2.2 Compatible Development Tools

The assembler works with many other Microchip tools, including:

- The MPLAB X IDE (www.microchip.com/mplab/mplab-x-ide) for all 8-bit PIC devices
- The MPLAB X Simulator
- The Command-line MDB Simulator—see the Microchip Debugger (MDB) User's Guide (DS52102)
- All Microchip debug tools and programmers (www.microchip.com/mplab/development-boards-and-tools)
- Demonstration boards and Starter kits that support 8-bit PIC devices

Check the tool's documentation to verify that it supports the device you plan to use.

3. Assembler Driver

The name of the command-line driver used by the MPLAB XC8 PIC Assembler is `pic-as`.

This driver can be invoked to perform both assembly and link steps and is the application called by development environments, such as the MPLAB X IDE, to build assembly projects.

The `pic-as` driver has the following basic command format:

```
pic-as [options] files [libraries]
```

Throughout this manual, it is assumed that the assembler applications are in your console's search path or that the full path is specified when executing the application.

It is customary to declare options (identified by a leading dash "-" or double dash "--") before the files' names; however, this is not mandatory.

The formats of the *options* are supplied in 3.4. Assembler Option Descriptions along with corresponding descriptions of the options' function.

The *files* can be an assortment of assembler source files and precompiled intermediate files. While the order in which these files are listed is not important, it can affect the allocation of code or data and can affect the names of some of the output files.

The *libraries* is a list of user-defined library files that will be searched by the assembler. The order of these files will determine the order in which they are searched. It is customary to insert the libraries after the list of source files; however, this is not mandatory.

3.1 Single-step Assembly

Assembly of one or more source files can be performed in just one step using the `pic-as` driver.

The following command will build both the assembly source files, passing these files to the appropriate internal applications, then link the generated code to form the final output.

```
pic-as -mcpu=16F877A main.S mdef.s
```

The driver will compile all source files, regardless of whether they have changed since the last build. Development environments (such as MPLAB® X IDE) and make utilities must be employed to achieve incremental builds (see 3.2. Multi-step Assembly).

Unless otherwise specified, a HEX file and ELF file are produced as the final output. The intermediate files remain after compilation has completed, but most other temporary files are deleted, unless you use the `-save-temps` option (see 3.4.28. Save-temps Option) which preserves all generated files. Note that some generated files can be in a different directory than your project source files (see also 3.4.27. O: Specify Output File).

3.2 Multi-step Assembly

A multi-step assembly method can be employed to achieve an incremental build of your project. Make utilities take note of which source files have changed since the last build and only rebuild these files to speed up assembly. From within MPLAB X IDE, you can select an incremental build (Build Project icon), or fully rebuild a project (Clean and Build Project icon).

Make utilities typically call the assembler multiple times: once for each source file to generate an intermediate file and once to perform the link step.

The option `-c` is used to create an intermediate file. This option stops after the assembler application has executed, and the resulting object output file will have a `.o` extension.

The intermediate files can then be specified to the driver during the second stage of compilation, when they will be passed to the linker.

The first two of the following command lines build an intermediate file for each assembly source file, then these intermediate files are passed to the driver again to be linked in the last command.

```
pic-as -mcpu=16F877A -c main.s
pic-as -mcpu=16F877A -c io.s
pic-as -mcpu=16F877A main.o io.o
```

As with any assembler, all the files that constitute the project must be present when performing the second (link) stage of compilation.

You might also wish to generate intermediate files to construct your own library files. See 7.1. Archiver/Librarian for more information on library creation.

3.3 Long Command Lines

The `pic-as` driver can be passed a command-line file containing driver options and arguments to circumvent any operating-system-imposed limitation on command line length.

A command file is specified by the `@` symbol, which should be immediately followed (i.e., no intermediate space character) by the name of the file containing the arguments. This same system of argument passing can be used by most of the internal applications called by the compiler driver.

Inside the file, each argument must be separated by one or more spaces and can extend over several lines when using a backslash-return sequence. The file can contain blank lines, which will be ignored.

The following is the content of a command file, `xyz.xc8` for example, that was constructed in a text editor and that contains the options and the file names required to compile a project.

```
-mcpu=18F26K40 -Wl,-Map=proj.map -Wa,-a \
main.S isr.S
```

After this file is saved, the compiler can be invoked with the following command:

```
pic-as @xyz.xc8
```

Command files can be used as a simple alternative to a make file and utility, and can conveniently store compiler options and source file names. The MPLAB X IDE also allows such files to be used. The file name is specified in the **XC8 Linker** > **Additional options** > **Use response file to link** field of the Project Properties.

3.4 Assembler Option Descriptions

Most aspects of the build process can be controlled using options passed to the assembler's command-line driver, `pic-as`.

All options are case sensitive and are identified by leading single or double dash character, e.g. `-o` or `--version`.

Use the `--help` option to obtain a brief description of accepted options on the command line.

If you are compiling from within the MPLAB X IDE, it will issue explicit options to the assembler that are based on the selections in the project's **Project Properties** dialog. The default project options might be different to the default options used by the assembler when running on the command line, so you should review your IDE properties to ensue that they are acceptable.

The summary of all available driver options tabulated below is followed by a detailed description of each option.

Table 3-1. PIC Assembler Driver Options

Option (Links to explanatory section)	**Controls**
`-c`	Generation of an intermediate object file

.....continued

Option (Links to explanatory section)	Controls
-Dmacro=text	Definition of preprocessor symbols
-dM	List all defined macros
-E	The generation of preprocessed-only files
--fill=options	Filling of unused memory
-fmax-errors	Number of errors before aborting
-gformat	The type of debugging information generated
-H	List included header files
--help	Display help information only
-Idir	Directories searched for headers
-llibrary	Which libraries are scanned
-Ldir	Directories searched for libraries
-mcallgraph=type	The type of callgraph printed in the map file
-mchecksum=specs	The generation and placement of a checksum or hash
-mcpu=device	The target device that code will be built for
-mdfp=path	Which device family pack to use
-m[no-]download	How the final HEX file is conditioned
-misa=set	Select PIC18 instruction set
-mmaxichip	Use of a hypothetical device with full memory
-mprint-devices	Chip information only
-mram=ranges	Data memory that is available for the program
-mreserve=ranges	What memory should be reserved
-mrom=ranges	Program memory that is available for the program
-mserial=options	The insertion of a serial number in the output
-msummary=types	What memory summary information is produced
-mwarn=level	The threshold at which warnings are output
-o file	Specify output file name
-save-temps	Whether intermediate files should be kept after compilation
-Umacro	The undefining of preprocessor symbols
-v	Verbose compilation output
--version	Display of assembler version information
-w	The suppression of all warning messages
-Wa,option	Options passed to the assembler
-Wp,option	Options passed to the preprocessor

.....continued	
Option (Links to explanatory section)	**Controls**
`-Wl,option`	Options passed to the linker
`-xlanguage`	The language in which the source files are interpreted
`-Xassembler option`	Options passed to the assembler
`-Xpreprocessor option`	Options passed to the preprocessor
`-Xlinker option`	System-specific options to passed to the linker

3.4.1 C: Compile To Intermediate File

The `-c` option is used to generate an intermediate file for each source file listed on the command line.

For assembly source files, compilation will terminate after executing the assembler, leaving behind relocatable object files with a `.o` extension.

This option is often used to facilitate multi-step builds using a make utility.

3.4.2 D: Define a Macro

The `-Dmacro=text` option allows you to define preprocessor macros. For macros to be subsequently processed, the source files must be preprocessed by having them use a `.S` file extension or by using the `-xassembler-wth-cpp` option.

A space may be present between the option and macro name.

With no `=text` specified in the option, this option defines a preprocessor macro called `macro` that will be considered to have been defined by any preprocessor directive that checks for such a definition (e.g. the `#ifdef` directive) and that will expand to be the value 1 if it is used in a context where it will be replaced. For example, when using the option, `-DMY_MACRO` (or `-D MY_MACRO`) and supplying the following code:

```
#ifdef MY_MACRO
movlw MY_MACRO;
#endif
```

the `movlw` instruction will be assembled and the value `1` will be assigned to the W register.

When the replacement, `text`, is specified with the option, the macro will subsequently expand to be the replacement specified with the option. So if the above example code was compiled with the option `-DMY_MACRO=0x55`, then the instruction would be assembled as: `movlw 0x55`

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

3.4.3 dM: Preprocessor Debugging Dumps Option

The `-dM` option has the preprocessor produce macro definitions that are in effect at the end of preprocessing. This option should be used in conjunction with the `-E` option, and if you want the output directed to a file, use also the `-o` option.

3.4.4 E: Preprocess Only

The `-E` option is used to generate preprocessed assembly source files (also called modules or translation units).

When this option is used, the build sequence will terminate after the preprocessing stage, leaving behind files with the same basename as the corresponding source file and with a `.i` extension.

You might check the preprocessed source files to ensure that preprocessor macros have expanded to what you think they should. The option can also be used to create assembly source files that do not require any separate header files. This is useful when sending files to a colleague or to obtain technical support without sending all the header files, which can reside in many directories.

3.4.5 Fill Option

The `--fill=`*`options`* option allows you to fill unused memory with specified values in a variety of ways.

This option is functionally identical to Hexmate's `-fill` option. For more detailed information and advanced controls that can be used with this option, refer to 7.2.2.12. Fill.

3.4.6 Max Errors

The `-fmax-errors=`*`n`* option sets the maximum number of errors each assembler application, as well as the driver, will display before execution is terminated.

By default, up to 20 error messages will be displayed by each application before the assembler terminates. The option `-fmax-errors=10`, for example, would ensure the applications terminate after only 10 errors.

3.4.7 G: Produce Debugging Information Option

The `-g`*`format`* option instructs the assembler to produce additional information, which can be used by hardware tools to debug your program.

The support formats are tabulated below.

Table 3-2. Supported Debugging File Formats

Format	Debugging file format
`-gcoff`	COFF
`-gdwarf-3`	ELF/DWARF release 3.
`-ginhx32`	Intel HEX with extended linear address records, allowing use of addresses beyond 64 KB.
`-ginhx032`	INHX32 with initialization of upper address to zero.

By default, the assembler produces DWARF release 3 files.

3.4.8 H: Print Header Files Option

The `-H` option prints to the console the name of each header file used, in addition to other normal activities.

3.4.9 Help

The `--help` option displays information on the `pic-as` assembler options, then the driver will terminate.

3.4.10 I: Specify Include File Search Path Option

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

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

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

3.4.11 L: Specify Library File Option

The `-l`*`library`* option looks for the specified file (with a `.a` extension) and scans this library archive for unresolved symbols when linking.

The only difference between using an `-l` option (e.g., `-lmylib`) and specifying a file name on the command line (e.g., `mylib.a`) is that the assembler will search for the library specified using `-l` in several directories, as specified by the `-L` option.

3.4.12 L: Specify Library Search Path Option

The `-Ldir` option allows you to specify an additional directory to be searched for library files that have been specified by using the `-l` option. The assembler will automatically search standard library locations, so you only need to use this option if you are linking in your own libraries.

3.4.13 Callgraph Option

The `-mcallgraph=type` option controls what sort of call graph is printed in the map file. The available types are shown in the table.

Table 3-3. Callgraph types

Type	Produces
`none`	No call graph
`crit`	Only critical paths in the callgraph
`std`	Standard, short-form call graph (default)
`full`	Full call graph

The callgraph is generated by the linker, primarily for the purposes of allocating memory to objects in the compiled stack. Those routines defining stack objects that are not overlaid with other stack objects and that are hence contributing to the program's data memory usage are considered as being on a critical path.

3.4.14 Checksum Option

The `-mchecksum=specs` option will calculate a hash value (for example checksum or CRC) over the address range specified and stores the result in the hex file at the indicated destination address. The general form of this option is as follows.

```
-mchecksum=start-end@destination[,specifications]
```

The *start*, *end* and *destination* attributes are, by default, hexadecimal constants. The addresses defining the input range (`start` - `end`) are typically made multiples of the algorithm width. If this is not the case, bytes (with value 0) will pad any missing input word locations. The `destination` is where the hash result will be stored. This address cannot be within the range of addresses over which the hash is calculated.

The following specifications are appended as a comma-separated list to this option.

Table 3-4. Checksum Arguments

Argument	Description
`width=n`	Optionally specifies the decimal width of the result. Results can be calculated for byte-widths of 1 to 4 bytes for most algorithms, but it represents the bit width for SHA algorithms. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order. This width argument is not required with any Fletcher algorithm, as they have fixed widths, but it may be used to alter the default endianism of the result.
`offset=nnnn`	Specifies an initial hexadecimal value or offset to be used in the hash calculation. It is not used with SHA algorithms.
`algorithm=n`	The decimal argument selects one of the hash algorithms implemented in Hexmate. The selectable algorithms are described in Table 3-5. If unspecified, the default algorithm used is 8-bit checksum addition (algorithm 1).
`polynomial=nn`	Selects the polynomial value when using CRC algorithms

.....continued	
Argument	**Description**
`code=nn.Base`	Selects a hexadecimal code that will trail each byte in the result. This can allow each byte of the hash result to be embedded within an instruction, for example `code=34` will embed each byte of the result in a `retlw` instruction on Mid-range devices, or `code=0000` will append two 0x00 bytes to each byte of the hash. This code sequence can be optionally followed by `.Base`, where `Base` is the number of bytes of the hash to be output before the trailing code sequence is appended. A specification of `code=1122.2`, for example, will output the bytes 0x11 and 0x22 after each two bytes of the hash result.
`revword=n`	Specifies a decimal word width. If this is non-zero, then bytes within each word are read in reverse order when calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 or 2. A zero width disables the reverse-byte feature, as if the `revword` suboption was not present. This suboption should be used when using Hexmate to match a CRC produced by a PIC hardware CRC module that use the Scanner module to stream data to it.
`skip=n.Bytes`	Specifies a decimal word width. If this is non-zero, then the MSB within each word is skipped for the purposes of calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 (which disables the skip feature, as if the `skip` suboption was not present) or greater than 1. This value can be optionally followed by `.Bytes`, where `Bytes` is a number representing the number of bytes to skip in each word, for example `skip=4.2` will skip the two most significant bytes in each 4-byte word.

Note that the reverse and skip features act on words that are aligned to the HEX file addresses, not to the position of the data in the sequence being processed. In other words, the alignment of the words are not affected by the `start` and `end` addresses specified with the option.

If an accompanying `--fill` option (3.4.5. Fill Option) has not been specified, unused locations within the specified address range will be automatically filled with 0xFFF for baseline devices, 0x3FFF for mid-range devices, or 0xFFFF for PIC18 devices. This is to remove any unknown values from the calculations and ensure the accuracy of the result.

For example:

```
-mchecksum=800-fff@20,width=1,algorithm=2
```

will calculate a 1-byte checksum from address 0x800 to 0xfff and store this at address 0x20. A 16-bit addition algorithm will be used. Table 3-5 shows the available algorithms and 7.2.3. Hash Value Calculations describes these in detail.

Table 3-5. Checksum Algorithm Selection

Selector	Algorithm description
-5	Reflected cyclic redundancy check (CRC)
-4	Subtraction of 32 bit values from initial value
-3	Subtraction of 24 bit values from initial value
-2	Subtraction of 16 bit values from initial value
-1	Subtraction of 8 bit values from initial value
1	Addition of 8 bit values from initial value
2	Addition of 16 bit values from initial value
3	Addition of 24 bit values from initial value
4	Addition of 32 bit values from initial value
5	Cyclic redundancy check (CRC)

.....continued	
Selector	**Algorithm description**
7	Fletcher's checksum (8 bit calculation, 2-byte result width)
8	Fletcher's checksum (16 bit calculation, 4-byte result width)
10	SHA-2 (currently only SHA256 is supported)

The hash calculations are performed by the Hexmate application. The information in this driver option is passed to the Hexmate application when it is executed.

3.4.15 Cpu Option

The `-mcpu=`*`device`* option must be used to specify the target device when building. This is the only option that is mandatory.

For example `-mcpu=18f6722` will select the PIC18F6722 device. To see a list of supported devices that can be used with this option, use the `-mprint-devices` option (3.4.20. Print-devices).

3.4.16 Dfp Option

The `-mdfp=`*`path`* option indicates that device-support for the target device (indicated by the `-mcpu` option) should be obtained from the contents of a Device Family Pack (DFP), where *`path`* is the path to the `xc8` sub-directory of the DFP.

When this option has not been used, the `pic-as` driver will where possible use the device-specific files provided in the assembler distribution.

The Microchip development environments automatically uses this option to inform the assembler of which device-specific information to use. Use this option on the command line if additional DFPs have been obtained for the assembler.

A DFP might contain such items as device-specific header files, configuration bit data and libraries, letting you take advantage of features on new devices without you having to otherwise update the assembler. DFPs never contain executables or provide bug fixes or improvements to any existing tools or standard library functions.

When using this option, the preprocessor will search for include files in the *`<DFP>`*`/xc8/pic/include/proc` and *`<DFP>`*`/xc8/pic/include` directories first, then search the standard search directories.

3.4.17 Download Option

The `-mdownload` option conditions the Intel HEX for use by bootloader. The `-mdownload-hex` option is equivalent in effect.

When used, this option will pad data records in the Intel HEX file to 16-byte lengths and will align them on 16-byte boundaries.

The default operation is to not modify the HEX file and this can be made explicit using the option `-mno-download`. (`-mno-download-hex`)

3.4.18 Isa Option

The `-misa=`*`set`* option allows the instruction set to be specified for PIC18 targets. The default is the standard PIC18 instruction set, a selection that can be made explicit by using `-misa=std`. Alternatively, the PIC18 extended instruction set can be selected by using `-misa=xinst`. When used with a device that does not support the requested instruction set, the option will be ignored and a warning issued.

Note that this option will permit the assembler to check for conformance of the assembly program to the selected instruction set, but it does not instruct the device to operate with the selected instruction set. Use the `XINST` configuration bit to enable this in your device.

3.4.19 Maxichip Option

The `-mmaxichip` option tells the assembler to build for a hypothetical device with the same physical core and peripherals as the selected device, but with the maximum allowable memory resources permitted by the device

family. You might use this option if your program does not fit in your intended target device and you wish to get an indication of the code or data size reductions needed to be able to program that device.

The assembler will normally terminate if the selected device runs out of program memory, data memory, or EEPROM. When using this option, the program memory of PIC18 and mid-range devices will be maximized to extend from address 0 to either the bottom of external memory or the maximum address permitted by the PC register, whichever is lower. The program memory of baseline parts is maximized from address 0 to the lower address of the Configuration Words.

The number of data memory banks is expanded to the maximum number of selectable banks as defined by the BSR register (for PIC18 devices), RP bits in the STATUS register (for mid-range devices), or the bank select bits in the FSR register (for baseline devices). The amount of RAM in each additional bank is equal to the size of the largest contiguous memory area within the physically implemented banks.

If present on the device, EEPROM is maximized to a size dictated by the number of bits in the EEADR or NVMADR register, as appropriate.

If required, check the map file (see 6.3. Map Files) to see the size and arrangement of the memory available when using this option with your device.

Note: When using the `-mmaxichip` option, you are not building for a real device. The generated code may not load or execute in simulators or the selected device. This option will not allow you to fit extra code into a device.

3.4.20 Print-devices

The `-mprint-devices` option displays a list of devices the assembler supports.

The names listed are those devices that can be used with the `-mcpu` option. This option will only show those devices that were officially supported when the assembler was released. Additional devices that might be available via device family packs (DFPs) will not be shown in this list.

The assembler will terminate after the device list has been printed.

3.4.21 Ram Option

The `-mram=`*`ranges`* option is used to adjust the data memory that is specified for the target device. Without this option, all the on-chip RAM implemented by the device is available, thus this option only needs be used if there are special memory requirements. Specifying additional memory that is not in the target device might result in a successful compilation, but can lead to code failures at runtime.

For example, to specify an additional range of memory to that already present on-chip, use:

```
-mram=default,+100-1ff
```

This will add the range from 100h to 1ffh to the on-chip memory. To only use an external range and ignore any on-chip memory, use:

```
-mram=0-ff
```

This option can also be used to reserve memory ranges already defined as on-chip memory in the relevant chipinfo file. To do this, supply a range prefixed with a minus character, `-`, for example:

```
-mram=default,-100-103
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 103h for allocation of RAM objects.

This option will adjust the memory ranges used by linker classes (see 6.1.1. A: Define Linker Class Option) . Any objects contained in a psect that do not use the classes affected by this option might be linked outside the valid memory specified by this option.

3.4.22 Reserve Option

The `-mreserve=`*`ranges`* option allows you to reserve memory normally used by the program. This option has the general form:

```
-mreserve=space@start:end
```

where *`space`* can be either of `ram` or `rom`, denoting the data and program memory spaces, respectively; and *`start`* and *`end`* are addresses, denoting the range to be excluded. For example, `-mreserve=ram@0x100:0x101` will reserve two bytes starting at address 100h from the data memory.

This option performs a similar task to the `-mram` and `-mrom` options, but it cannot be used to add additional memory to that available for the program.

3.4.23 Rom Option

The `-mrom=`*`ranges`* option is used to change the default program memory that is specified for the target device. Without this option, all the on-chip program memory implemented by the device is available, thus this option only needs be used if there are special memory requirements. Specifying additional memory that is not in the target device might result in a successful compilation, but can lead to code failures at runtime.

For example, to specify an additional range of memory to that on-chip, use:

```
-mrom=default,+100-2ff
```

This will add the range from 100h to 2ffh to the on-chip memory. To only use an external range and ignore any on-chip memory, use:

```
-mrom=100-2ff
```

This option can also be used to reserve memory ranges already defined as on-chip memory in the chip configuration file. To do this supply a range prefixed with a minus character, `-`, for example:

```
-mrom=default,-100-1ff
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 1ffh for allocation of ROM objects.

This option will adjust the memory ranges used by linker classes (see 6.1.1. A: Define Linker Class Option) . Any code or objects contained in a psect that do not use the classes affected by this option might be linked outside the valid memory specified by this option.

Note that some psects must be linked above a threshold address, most notably some psects that hold `const`-qualified data. Using this option to remove the upper memory ranges can make it impossible to place these psects.

3.4.24 Serial Option

The `-mserial=`*`options`* option allows a hexadecimal code to be stored at a particular address in program memory. A typical task for this option might be to position a serial number in program memory.

The byte-width of data to store is determined by the byte-width of the hexcode parameter in the option. For example, to store a one-byte value, 0, at program memory address 1000h, use `-mserial=00@1000`. To store the same value as a four byte quantity use `-mserial=00000000@1000`.

This option is functionally identical to Hexmate's `-serial` option. For more detailed information and advanced controls that can be used with this option (refer to 7.2.2.21. Serial).

3.4.25 Summary Option

The `-msummary=`*`type`* option selects the type of information that is included in the summary that is displayed once the build is complete. By default, or if the `mem` type is selected, a memory summary with the total memory usage for all memory spaces is shown.

A psect summary can be shown by enabling the `psect` type. This shows individual psects after they have been grouped by the linker and the memory ranges they cover. Table 4-20 shows what summary types are available. The default output printed corresponds to the `mem` setting.

SHA hashes for the generated hex file can also be shown using this option. These can be used to quickly determine if anything in the hex file has changed from a previous build.

Table 3-6. Summary Types

Type	Shows
`psect`	A summary of psect names and the addresses where they were linked will be shown.
`mem`	A concise summary of memory used will be shown (default).
`class`	A summary of all classes in each memory space will be shown.
`hex`	A summary of addresses and HEX files that make up the final output file will be shown.
`file`	Summary information will be shown on screen and saved to a file.
`sha1`	A SHA1 hash for the hex file.
`sha256`	A SHA256 hash for the hex file.
`xml`	Summary information will be shown on the screen, and usage information for the main memory spaces will be saved in an XML file.
`xmlfull`	Summary information will be shown on the screen, and usage information for all memory spaces will be saved in an XML file.

If specified, the XML files contain information about memory spaces on the selected device, consisting of the space's name, addressable unit, size, amount used and amount free.

3.4.26 Warn Option

The `-mwarn=`*`level`* option is used to set the warning level threshold. Allowable warning levels range from `-9` to `9`. The warning level determines how pedantic the assembler is about dubious type conversions and constructs. Each warning has a designated warning level; the higher the warning level, the more important the warning message. If the warning message's warning level exceeds the set threshold, the warning is printed. The default warning level threshold is `0` and will allow all normal warning messages.

Use this option with care as some warning messages indicate code that is likely to fail during execution, or compromise portability.

3.4.27 O: Specify Output File

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

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

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

Only the base name of the file specified with this option has significance. You cannot use this option to change the extension of an output file.

3.4.28 Save-temps Option

The `-save-temps` option instructs the assembler to keep temporary files after compilation has finished.

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

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

The `-save-temps=obj` form of this option is similar to `-save-temps`, but if the `-o` option is specified, the temporary files are placed in the same directory as the output object file. If the `-o` option is not specified, the `-save-temps=obj` switch behaves like `-save-temps`.

3.4.29 U: Undefine Macros

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

Any macro defined using `-D` will be undefined by this option. All `-U` options are evaluated after all `-D` options.

3.4.30 V: Verbose Compilation

The `-v` option specifies verbose compilation.

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

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

3.4.31 Version

The `--version` option prints assembler version information then exits.

3.4.32 W: Disable all Warnings Option

The `-w` option inhibits all warning messages, and thus should be used with caution.

3.4.33 Wa: Pass Option To The Assembler, Option

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

3.4.34 Wp: Pass Option To The Preprocessor Option

The `-Wp,`*`option`* option passes *`option`* to the preprocessor, where it will be interpreted as a preprocessor option. If option contains commas, it is split into multiple options at the commas.

3.4.35 Wl: Pass Option To The Linker, Option

The `-Wl,`*`option`* option passes *`option`* to the linker application where it will be interpreted as a linker option. If *`option`* contains commas, it is split into multiple options at the commas. This means that this option can not be used to pass in a linker option such as `-pcodeStart,codeEnd`, which uses the comma to separate psect names. In this case, use the `-Xlinker` option.

3.4.36 X: Specify Source Language Option

The `-x`*`language`* option allows you to specify that the source files that follow are written in the specified language, regardless of the extension they use.

The languages allowed by the assembler are tabulated below.

Table 3-7. Language options

Language	Description
`assembler`	Assembly source code
`assembler-with-cpp`	Assembly source code that must be preprocessed

For example, the command:

```
pic-as -mcpu=18f4520 -c -xassembler-with-cpp init.s
```

will tell the assembler to run the preprocessor over the assembly source file, even though the `init.s` file name does not use a `.S` extension.

3.4.37 Xassembler Option

The `-Xassembler` *`option`* option passes *`option`* to the assembler, where it will be interpreted as an assembler option. You can use this to supply system-specific assembler options that the assembler does not know how to recognize or that can't be parsed by the `-Wa` option.

3.4.38 Xpreprocessor Option

The `-Xpreprocessor` *`option`* option passes *`option`* to the preprocessor, where it will be interpreted as a preprocessor option. You can use this to supply system-specific preprocessor options that the assembler does not know how to recognize.

3.4.39 Xlinker Option

The `-Xlinker` *`option`* option pass *`option`* to the linker where it will be interpreted as a linker option. You can use this to supply system-specific linker options that the assembler does not know how to recognize.

3.5 MPLAB X IDE Integration

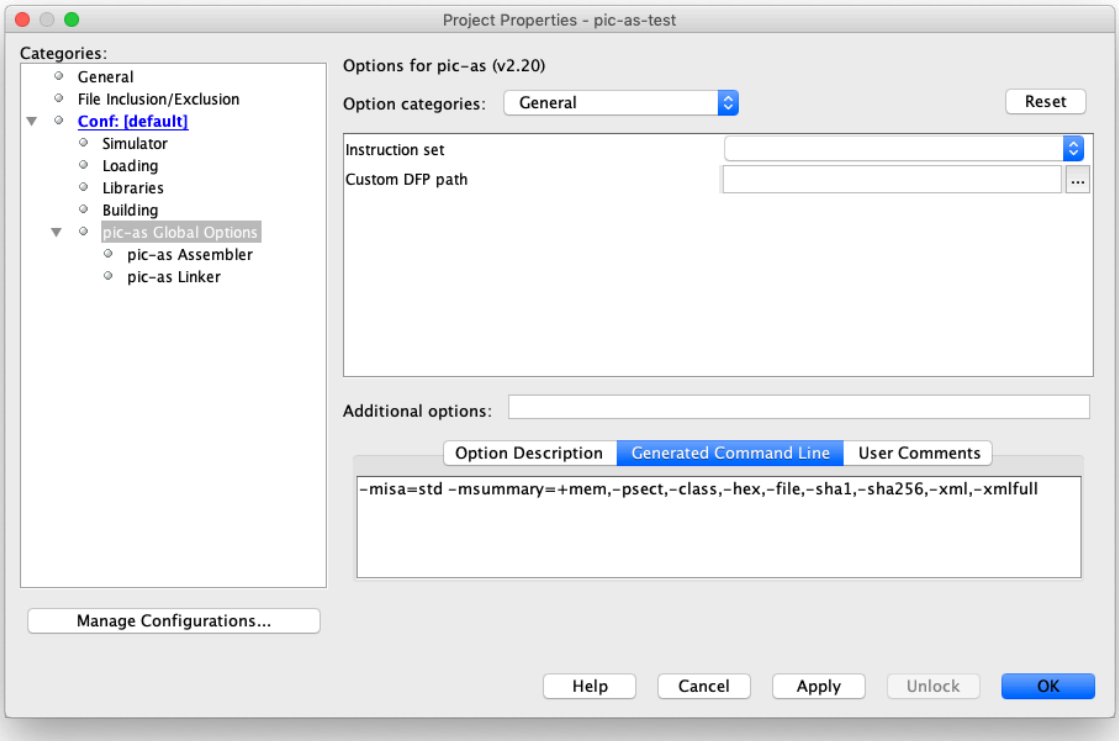
The 8-bit language tools may be integrated into and controlled from the MPLAB X IDE, to provide a GUI-based development of application code for the 8-bit PIC MCU families of devices.

For installation of the IDE, and the creation and setup of projects to use the MPLAB XC8 PIC Assembler, see the *MPLAB® X IDE User's Guide*.

3.5.1 MPLAB X IDE Option Equivalents

The following descriptions map the MPLAB X IDE's **Project Properties** controls to the MPLAB XC8 command-line driver options. Reference is given to the relevant section in the user's guide to learn more about the option's function. In the IDE, click any option to see online help and examples shown in the **Option Description** field in the lower part of the dialog.

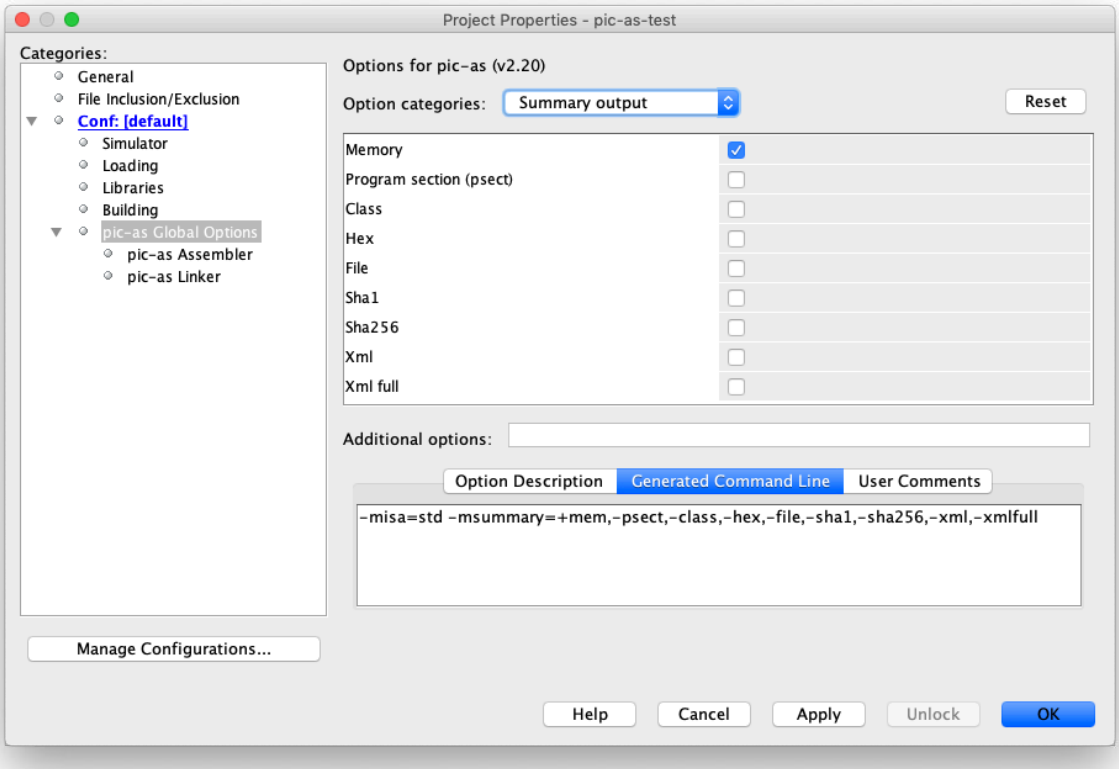
3.5.2 PIC Assembler Global Options - General



Instruction set This selector allows you to specify whether the Standard or Extended instruction set should be used for projects targeting PIC18 devices. See 3.4.18. Isa Option.

Custom DFP path If you need to use an alternative Device Family Pack (DFP), enter the path to the DFP you wish to use in this field. See 3.4.16. Dfp Option.

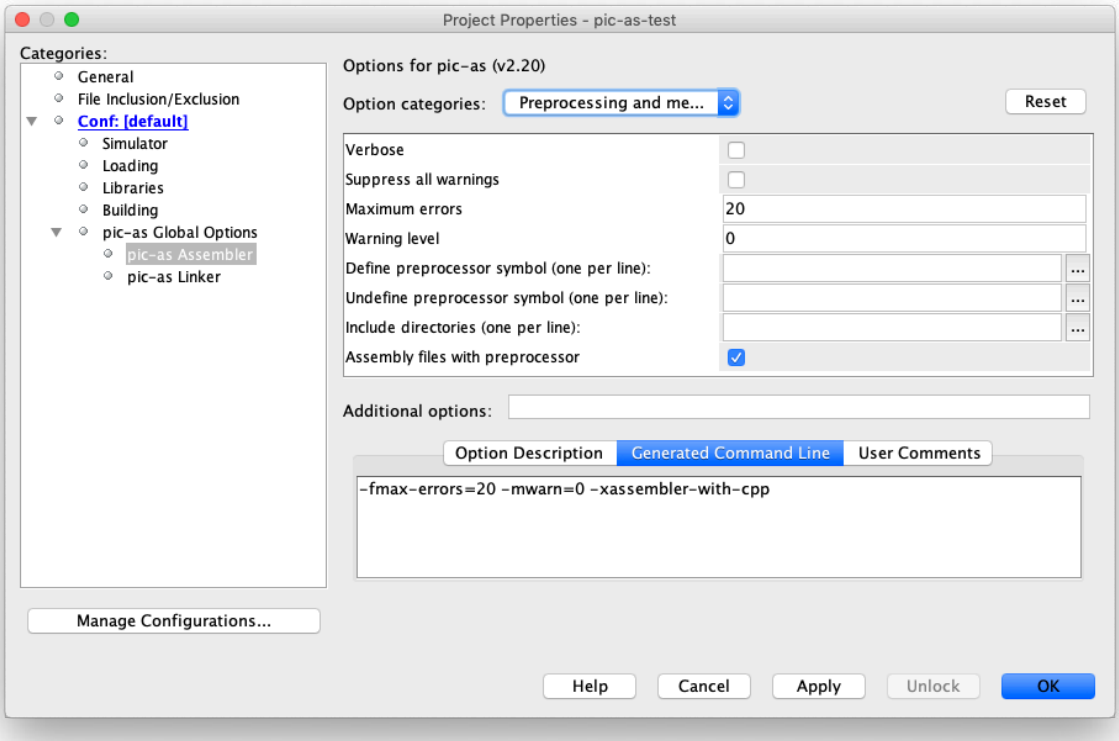
3.5.3 PIC Assembler Global Options - Summary



Summary selectors

The checkboxes in this dialog selects the type of information that is included in the summary that is displayed in the **Output** window once the build is complete. See 3.4.25. Summary Option option for full details of each summary.

3.5.4 PIC-AS Assembler - Preprocessing and messaging



Verbose — Selecting this checkbox will perform verbose compilation, showing the command lines issued to the internal applications. See 3.4.30. V: Verbose Compilation.

Suppress all warnings — Selecting this checkbox will prevent all warning messages from being issued. See 3.4.32. W: Disable all Warnings Option.

Maximum errors — Use this field to specify the maximum number of errors that will be generated before the assembler exits. See 3.4.6. Max Errors.

Warning level — Use this field to specify the warning level threshold. Only warnings with higher levels will be emitted during a build. See 3.4.26. Warn Option.

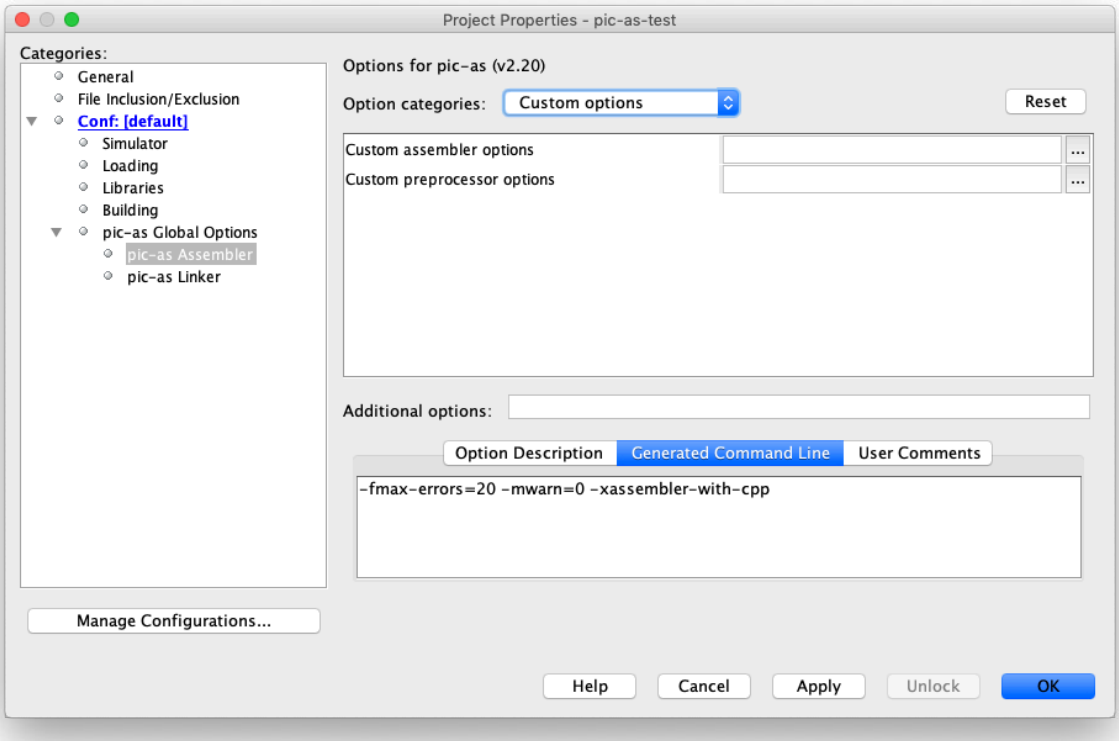
Define preprocessor symbol — Use this field to define preprocessor macros. See 3.4.2. D: Define a Macro.

Undefine preprocessor symbol — Use this field to undefine preprocessor macros. See 3.4.29. U: Undefine Macros.

Include directories — Use this field to specify the directories searched for header files. See 3.4.10. I: Specify Include File Search Path Option.

Assembler files with preprocessor — Select this checkbox to have assembly source files preprocessed before being passed to the assembler, regardless of the file extension. See 3.4.36. X: Specify Source Language Option.

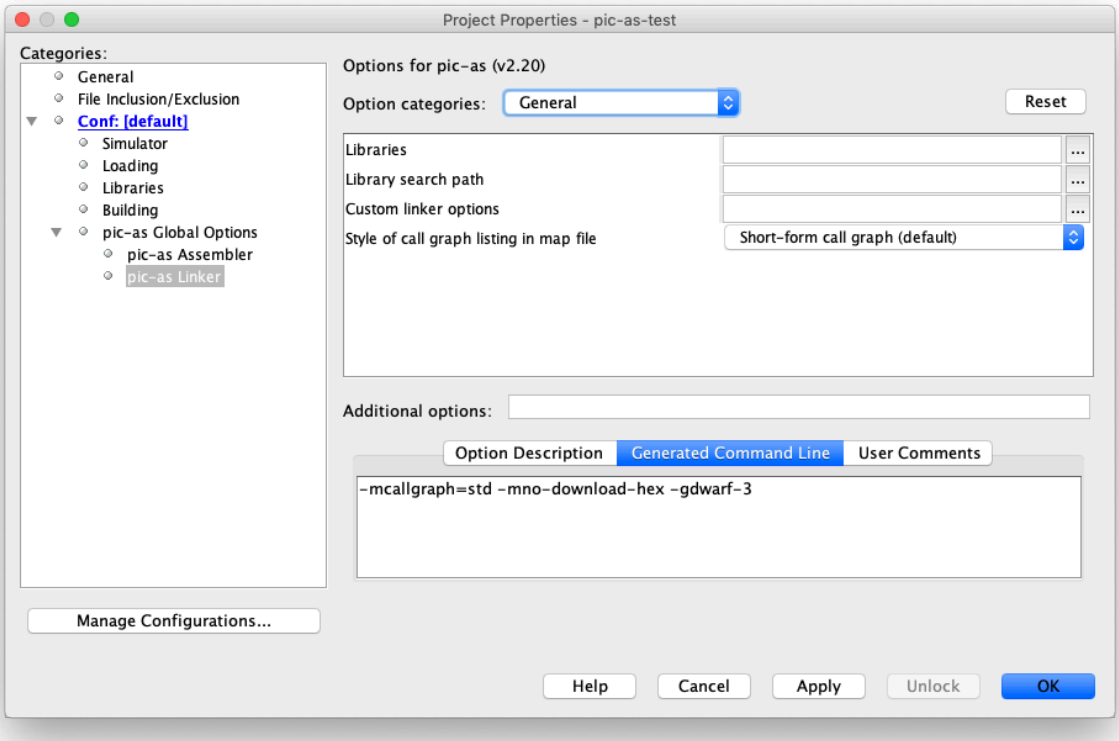
3.5.5 PIC-AS Assembler - Custom options



Custom assembler options Use this field to have options not directly available in the project properties dialog passed to the assembler. See 3.4.33. Wa: Pass Option To The Assembler, Option.

Custom preprocessor options Use this field to have options not directly available in the project properties dialog passed to the preprocessor. See 3.4.34. Wp: Pass Option To The Preprocessor Option.

3.5.6 PIC-AS Linker - General



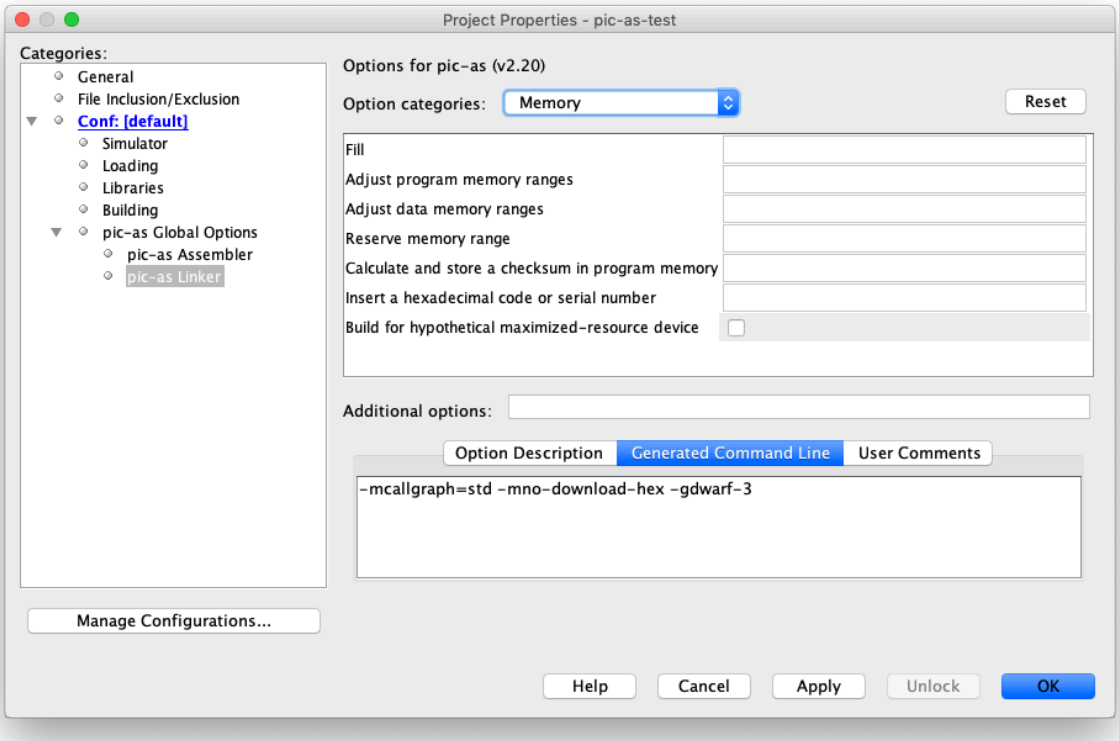
Libraries Use this field to add the names of library archives. See 3.4.11. L: Specify Library File Option.

Library search path Use this field to specify the paths to be searched for library archive files. See 3.4.12. L: Specify Library Search Path Option.

Custom linker options Use this field to have options not directly available in the project properties dialog passed to the linker. See 3.4.35. Wl: Pass Option To The Linker, Option.

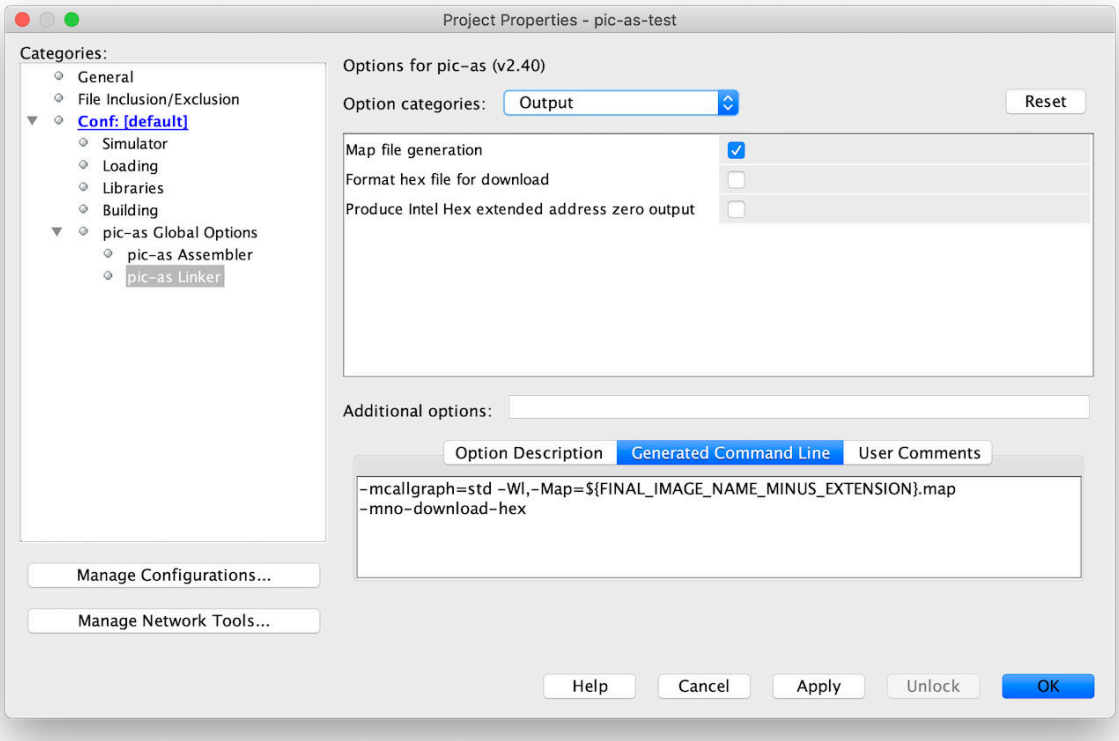
Style of call graph listing in map file Use this selector to choose what level of detail you require in the call graph printed in the map file. See 3.4.13. Callgraph Option.

3.5.7 PIC-AS Linker - Memory



Fill	Use this field to specify the command to fill unused memory. See 3.4.5. Fill Option.
Adjust program memory ranges	Use this field to adjust the program memory available on the device. See 3.4.23. Rom Option.
Adjust data memory ranges	Use this field to adjust the data memory available on the device. See 3.4.21. Ram Option.
Reserve memory range	Use this field to reserve space in program and data memory. See 3.4.22. Reserve Option.
Calculate and store a checksum in program memory	Use this field to specify a command to have a hash value calculated over the program output and stored in the HEX file. See 3.4.14. Checksum Option.
Insert a hexadecimal code or serial number	Use this field to specify a command to have a code stored in the HEX file. See 3.4.24. Serial Option.
Build for a hypothetical maximized-resource device	Select this checkbox to build for a hypothetical device with the same physical core and peripherals as the selected device, but with the maximum allowable memory resources permitted by the device family. See 3.4.19. Maxichip Option.

3.5.8 PIC-AS Linker - Output



Map file generation — Select this checkbox to have a map file generated by the linker. This will issue the appropriate option to the linker using the driver's `-Wl,` option (see 3.4.35. Wl: Pass Option To The Linker, Option).

Format hex file for download — This checkbox controls the special formatting of the final HEX file. See 3.4.17. Download Option.

Produce Intel Hex extended address zero output — Select this checkbox to have an extended address Intel HEX file produced. See 3.4.7. G: Produce Debugging Information Option.

4. MPLAB XC8 Assembly Language

Information about the source language accepted by the macro assemblers is described in this section.

All opcode mnemonics and operand syntax are specific to the target device, and you should consult your device data sheet. Additional mnemonics, deviations from the instruction set, and assembler directives are documented in this section.

4.1 Assembly Instruction Deviations

The MPLAB XC8 assembler can use a slightly modified form of assembly language to that specified by the Microchip data sheets. This form is generally easier to read, but the form specified on the data sheet can also be used. The following information details allowable deviations to the instruction format as well as pseudo instructions that can be used in addition to the device instruction set.

4.1.1 Destination And Access Operands

To specify the destination for byte-orientated file register instructions, you may use the operands from either style shown in the table below. The wreg destination indicates that the instruction result will be written to the W register and the file register destination indicates that the result will be written to the register specified by the instruction's file register operand. This operand is usually represented by `,d` in the device data sheet.

Table 4-1. Destination Operand Styles

Style	Wreg destination	File register destination
XC8	`,w`	`,f`
MPASM	`,0`	`,1`

For example (ignoring bank selection and address masking for this example):

```
addwf   foo,w        ;add wreg to foo, leaving the result in wreg
addwf   foo,f        ;add wreg to foo, updating the content of foo
addwf   foo,0        ;add wreg to foo, leaving the result in wreg
addwf   foo,1        ;add wreg to foo, updating the content of foo
```

It is highly recommended that the destination operand is always specified with those instructions where it is needed. If the destination operand is omitted, the destination is assumed to be the file register.

To specify the RAM access bit for PIC18 devices, you may use operands from either style shown in the table below. Banked access indicates that the file register address specified in the instruction is just an offset into the currently selected bank. Unbanked access indicates that the file register address is an offset into the Access bank, or common memory.

Table 4-2. RAM Access Operand Styles

Style	Banked access	Unbanked access
XC8	`,b`	`,c` or `,a`
MPASM	`,1`	`,0`

This operand is usually represented by `,a` in the device data sheet.

Alternatively, an instruction operand can be preceded by the characters "`c:`" to indicate that the address resides in the Access bank. For example:

```
addwf   bar,f,c         ;add wreg to bar in common memory
addwf   bar,f,a         ;add wreg to bar in common memory
addwf   bar,1,0         ;add wreg to bar in common memory
addwf   bar,f,b         ;add wreg to bar in banked memory
```

```
addwf   bar,1,1         ;add wreg to bar in banked memory
btfsc   c:bar,3            ;test bit three in the common memory symbol bar
```

It is recommended that you always specify the RAM access operand or the common memory prefix. If these are not present, the instruction address is absolute and the address is within the upper half of the access bank (which dictates that the address must not masked), the instruction will use the access bank RAM. In all other situations, the instruction will access banked memory.

If you use the XC8 style, the destination operand and the RAM access operand can be listed in any order for PIC18 instructions. For example, the following two instructions are identical:

```
addwf   foo,f,c
addwf   foo,c,f
```

Always be consistent in the use of operand style for each instruction, and preferably, that style should remain consistent through the program. For example, the instruction `addwf bar,1,c` (which uses the MPASM-style `,1` destination suffix and the XC8-style `,c` unbanked access suffix together in the same instruction) is illegal.

For example, the following instructions show the W register being moved to first, an absolute location; and then to an address represented by an identifier. Bank selection and masking has been used in this example. The PIC18 opcodes for these instructions, assuming that the address assigned to `foo` is 0x516 and to `bar` is 0x55, are shown below.

```
6EE5  movwf 0FE5h             ;write to access bank location 0xFE5
6E55  movwf bar,c             ;write to access bank location 0x55
0105  BANKSEL(foo)            ;set up BSR to access foo
6F16  movwf BANKMASK(foo),b   ;write to foo (banked)
6F16  movwf BANKMASK(foo)     ;defaults to banked access
```

Notice that the first two instruction opcodes have the RAM access bit (bit 8 of the op-code) cleared, but that the bit is set in the last two instructions.

Note: The manual masking of addresses used as instruction operands (for example using the `BANKMASK()` or `PAGEMASK()` macros or ANDing the operand with a mask) is not necessary when using the `-Wl,--fixupoverflow` option and any of the `ignore`, `warn`, or `lstwarn` arguments.

4.1.2 Bank And Page Selection

The `BANKSEL()` pseudo instruction can be used to generate instructions to select the bank of the operand specified. The operand should be the symbol or address of an object that resides in the data memory.

Depending on the target device, the generated code will either contain one or more bit instructions to set/clear bits in the appropriate register, or use a `movlb` instruction (in the case of enhanced mid-range or PIC18 devices). As this pseudo instruction can expand to more than one instruction on mid-range or baseline parts, it should not immediately follow a `btfsx` instruction on those devices. For example:

```
movlw 20
BANKSEL(_foobar)    ;select bank for next file instruction
movwf BANKMASK(_foobar)  ;write data and mask address
```

Note: The manual masking of addresses used as instruction operands (for example using the `BANKMASK()` or `PAGEMASK()` macros or ANDing the operand with a mask) is not necessary when using the `-Wl,--fixupoverflow` option and any of the `ignore`, `warn`, or `lstwarn` arguments.

In the same way, the `PAGESEL()` pseudo instruction can be used to generate code to select the page of the address operand. For the current page, you can use the location counter, `$`, as the operand.

Depending on the target device, the generated code will either contain one or more instructions to set/clear bits in the appropriate register, or use a `movlp` instruction in the case of enhanced mid-range PIC devices. As the directive could expand to more than one instruction, it should not immediately follow a `btfsx` instruction. For example:

```
fcall _getInput
PAGESEL $        ;select this page
```

This directive is accepted when compiling for PIC18 targets but has no effect and does not generate any code. Support is purely to allow easy migration across the 8-bit devices.

4.1.3 Address Masking

All assembly identifiers represent a full address. The upper bits of a file register address represent the bank number of the object. Similarly, the upper bits of a program memory address represent the page number of the destination label. Such addresses might not be immediately usable as operands to some PIC instructions and might need to be masked or truncated to fit within the instruction's address field.

There are two ways of handling addresses when they are used as an instruction operand. The first is to have the linker automatically truncate these addresses to suite the instruction. The second is to manually mask the bank or page bits from the address. Both approaches have advantages. Automatic truncation is by far the easiest to use and does not clutter your assembly source with ancillary expressions in instruction operands, but this truncation is applied to all operands of all instructions in your program, and it might hide genuine errors that would otherwise be issued. Masking operands manually must be performed whenever needed but can be done in such a way that it confirms assumptions you have made regarding the location of the object, increasing the reliability of your program.

To have the linker automatically truncate all addresses to fit the instruction, use the `-Wl,--fixupoverflow=`*`action`* option and one of the `warn`, `lstwarn`, or `ignore` action arguments. See 6.1.31. Fixupoverflow Linker Option for more information on this option. Note that this option (except when an action argument of `error` is used) will suppress fixup overflow errors associated with all instructions in your program. It is recommended that you select an action argument of `warn` or `lstwarn` so that the generated warnings will help you confirm that there is no potential for program failure. If you are using this option to suppress fixup overflow errors, you do not need to mask the addresses used with any instructions, for example:

```
BANKSEL  flags            ;select data bank of flags
subwf    flags            ;use flags without masking its address
PAGESEL  myFunc           ;select the page of myFunc
call     myFunc           ;use myFunc without masking its address
```

To manually mask a file register address (data memory), use the `BANKMASK()` macro. To manually mask a program address used with the `call` and `goto` flow control instructions, use the `PAGEMASK()` macro. Both these macros AND out the bank or page information in the address using a suitable device-specific mask. They are available once you include `<xc.inc>` into a source module. Use of these macros (rather than manually using the AND operator, `&`) increases code portability across Microchip devices, since they adjust the mask to suit the bank or page size of the target device. The following code masks the addresses used by the `subwf` and `call` instructions:

```
BANKSEL  flags            ;select data bank of flags
subwf    BANKMASK(flags)  ;remove bank bits from address to prevent fixup overflow
PAGESEL  myFunc           ;select the page of myFunc
call     PAGEMASK(myFunc) ;remove page bits from address to prevent fixup overflow
```

Rather than ANDing out the bank information in an address using either the `BANKMASK()` or `PAGEMASK()` macros, the address can be XORed (`^` operator) with a bitmask that represents the expected bank or page bits in the address. If the address falls in the bank or page that was expected, then the upper bits of the bitmask and address will XOR to zero; if this is not the case, the XOR will produce a non-zero component in the upper bits of the address and trigger a fixup overflow error from the linker (assuming this error has not been disabled by the `-Wl,--fixupoverflow` option).

The following example for a 16Fxxxx device (which has a data bank size of 0x80) selects bank 2 before accessing the symbol `flags`. The `flags` symbol has been XORed with the mask 0x100, which represents the bitmap of bank 2 addresses with the address offset zeroed.

```
movlb  2    ;select bank 2
subwf  flags^0x100
```

If `flags` was linked at address 0x153 (a bank 2 address), then 0x153^0x100 will result in the value 0x53, which fits into the address field of the `subwf` instruction. If `flags` was accidentally linked to address 0x34 (a bank 0 address), then 0x34^0x100 yields the result 0x134, which is too large to fit into the 7-bit wide address field of the `subwf` instruction and which will trigger a fixup overflow error, alerting you to the problem.

Do not use the `BANKMASK()` or `PAGESEL()` macros with any instruction that expects its operand to be a full address, such as the PIC18's `movff` instruction for example.

Note that address masking is a fundamentally different operation to bank or page selection. Neither the `-Wl,--fixupoverflow` option nor the `BANKMASK()` or `PAGEMASK()` macros select the bank or page of the object being accessed, called, or jumped to. Regardless of how you handle address masking, you must always ensure that your program contains the instructions to select the correct bank or page when required, as described in 4.1.2. Bank And Page Selection. The one exception is when you use the `fcall` and/or `ljmp` pseudo instructions (see 4.1.7. Long Jumps And Calls), which perform both page selection and address masking for you.

4.1.4 Movfw Pseudo Instruction

The `movfw` pseudo instruction implemented by MPASM is not implemented in the MPLAB XC8 assemblers. You will need to use the standard PIC instruction that performs an identical function. Note that the MPASM instruction:

```
movfw foobar
```

maps directly to the standard PIC instruction:

```
movf foobar,w
```

4.1.5 Movff/movffl Instructions

The `movff` instruction is a physical device instruction, but for PIC18 devices that have extended data memory, it also serves as a placeholder for the `movffl` instruction.

For these devices, when generating output for the `movff` instruction, the assembler checks the psects that hold the operand symbols. If the psect containing the source operand and the psect containing the destination operand both specify the `lowdata` psect flag, the instruction is encoded as the two-word `movff` instruction. If an operand is an absolute address and that address is in the lower 4 KB of memory, then that is also considered acceptable for the shorter form of the instruction. In all other situations, the instruction is encoded as a three-word `movffl` instruction.

Note that assembly list files will always show the `movff` mnemonic, regardless of how it is encoded. Check the number of op-code words to determine which instruction was encoded.

4.1.6 Interrupt Return Mode

The `retfie` PIC18 instruction can be followed by "`f`" (no comma) to indicate that the shadow registers should be retrieved and copied to their corresponding registers on execution. Without this modifier, the registers are not updated from the shadow registers. This syntax is not relevant for Baseline and Mid-range devices.

The following examples show both forms and the opcodes they generate.

```
0011  retfie f    ;shadow registers copied
0010  retfie      ;return without copy
```

The "`0`" and "`1`" operands indicated in the device data sheet can be alternatively used if desired.

4.1.7 Long Jumps And Calls

The assembler recognizes several mnemonics that expand into regular PIC MCU assembly instructions. The mnemonics are `fcall` and `ljmp`.

On baseline and mid-range parts, these instructions expand into regular `call` and `goto` instructions respectively, but also ensure the instructions necessary to set the bits in PCLATH (for mid-range devices) or STATUS (for baseline devices) will be generated, should the destination be in another page of program memory. Page selection instructions can appear immediately before the `call` or `goto`, or be generated as part of, and immediately after, a previous `fcall`/`ljmp` mnemonic.

On PIC18 devices, these mnemonics are present purely for compatibility with smaller 8-bit devices and are always expanded as regular PIC18 `call` and `goto` instructions.

These special mnemonics should be used where possible, as they make assembly code independent of the final position of the routines that are to be executed.

The operand to the `fcall` and `ljmp` mnemonics should not be masked, regardless of target device. The full address is required to determine the destination address and page (where applicable). When the mnemonic is expanded, the address used with either the `call` or `goto` instruction in the expansion will be automatically masked. When using a `call` or `goto` instruction directly in your source code, always apply the appropriate mask to the operand.

The following mid-range PIC example shows an `fcall` instruction in the assembly list file. You can see that the `fcall` instruction has expanded to five instructions. In this example, there are two bit instructions that set/clear bits in the PCLATH register. Bits are also set/cleared in this register after the call to reselect the page that was selected before the `fcall`.

```
13  0079  3021                      movlw   33
14  007A  120A  158A  2000          fcall   _phantom
          120A  118A
15  007F  3400                      retlw   0
```

Since `fcall` and `ljmp` instructions can expand into more than one instruction, they should never be preceded by an instruction that can skip, e.g., a `btfsc` instruction.

The `fcall` and `ljmp` instructions assume that the psect that contains them is smaller than a page. Do not use these instructions to transfer control to a label in the current psect if it is larger than a page. The default linker options will not permit code psects to be larger than a page.

On PIC18 devices, the regular `call` instruction can be followed by a "`,f`" to indicate that the W, STATUS and BSR registers should be pushed to their respective shadow registers. This replaces the "`,1`" syntax indicated on the device data sheet.

4.1.8 Relative Branches

The PIC18 devices implement conditional relative branch instructions, e.g., `bz`, `bnz`. These instructions have a limited jump range compared to the `goto` instruction.

Unlike the MPLAB XC8 C Compiler, the PIC Assembler will never transform relative branch sequences to increase their range. If you need a relative branch that can reach targets outside the usual instruction range, use a relative branch with the reverse condition over a `goto` instruction. For example, instead of writing:

```
  bz next
```

write something like:

```
  bnz tmp
  goto next
tmp:
```

4.2 Statement Formats

Valid statement formats are shown in Table 4-3.

The *`label`* field is optional and, if present, should contain one identifier. A label can appear on a line of its own, or precede a mnemonic as shown in the second format.

The third format is only legal with certain assembler directives, such as `MACRO`, `SET` and `EQU`. The *`name`* field is mandatory and should contain one identifier.

If the assembly source file is first processed by the preprocessor, then it can also contain lines that form valid preprocessor directives. See 5.1. Preprocessor Directives for more information on the format for these directives.

There is no limitation on what column or part of the line in which any part of the statement should appear.

Table 4-3. Assembler Statement Formats

Format #	Field1	Field2	Field3	Field4
Format 1	*`label`*`:`			

.....continued				
Format #	**Field1**	**Field2**	**Field3**	**Field4**
Format 2	*label*:	*mnemonic*	*operands*	*; comment*
Format 3	name	*pseudo-op*	*operands*	*; comment*
Format 4	*; comment only*			
Format 5	empty line			

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 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 or use the `and` form of this operator. In a macro argument list, the angle brackets < and > are used to quote macro arguments.

4.4 Comments

An assembly comment is initiated with a semicolon that is not part of a string or character constant, for example:

```
movlw 22   ;this value will ensure there is a good safety margin
```

If the assembly file is first processed by the C preprocessor, then the file can also contain C or C++ style comments using the standard `/* ... */` and `//` syntax.

Avoid using assembly comments (`;comment`) in preprocessor directives, especially the `#define` directive. Assembly comments are not removed by the C preprocessor prior to macro substitution and so will appear in the substituted text, possibly resulting in build errors. Always use C or C++ style comments in these situations.

4.5 Constants

4.5.1 Numeric Constants

The assembler performs all arithmetic with signed 32-bit precision.

The default radix for all numbers is 10. Other radices can be specified by a trailing base specifier, as given in the following table.

Table 4-4. Numbers And Bases

Radix	Format
Binary	Digits 0 and 1 followed by `B`.
Octal	Digits 0 to 7 followed by `O`, `Q`, `o` or `q`.
Decimal	Digits 0 to 9 followed by `D`, `d` or nothing.
Hexadecimal	Digits 0 to 9, A to F preceded by `0x` or followed by `H` or `h`.

Hexadecimal numbers must have a leading digit (e.g., `0ffffh`) to differentiate them from identifiers. Hexadecimal digits are accepted in either upper or lower case.

The binary digits suffix (B) must be in upper case.

In expressions, real numbers are accepted in the usual format, and are interpreted as IEEE 32-bit format.

4.5.2 Character Constants And Strings

A character constant is a single character enclosed in single quotes '.

Multi-character constants, or strings, are a sequence of characters, not including carriage return or newline characters, enclosed within matching quotes. Either single quotes ' or double quotes " can be used, but the opening and closing quotes must be the same.

4.6 Identifiers

Assembly identifiers are user-defined symbols representing memory locations or numbers. A symbol can contain any number of characters drawn from alphabetics, numerics, as well as special characters: dollar, $; question mark, ?; and underscore, _.

The first character of an identifier cannot be numeric nor the $ character. The case of alphabetics is significant, e.g., Fred is not the same symbol as fred. Some examples of identifiers are shown here:

```
An_identifier
an_identifier
an_identifier1
?$_12345
```

An identifier cannot have the same symbol (any case) as any of the assembly code mnemonics (e.g. movlw or return) assembler directives (e.g. SET or LIST), directive argument tokens (e.g. hex or push), or operators (e.g. mod or nul).

4.6.1 Significance Of Identifiers

Users of other assemblers that attempt to implement forms of data typing for identifiers should note that this assembler attaches no significance to any symbol, and places no restrictions or expectations on the usage of a symbol.

The names of psects (program sections) and ordinary symbols occupy separate, overlapping name spaces, but other than this, the assembler does not care whether a symbol is used to represent bytes, words or sports cars. No special syntax is needed or provided to define the addresses of bits or any other data type, nor will the assembler issue any warnings if a symbol is used in more than one context. The instruction and addressing mode syntax provide all the information necessary for the assembler to generate correct code.

4.6.2 Assembler-generated Identifiers

Where a LOCAL directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form ??*nnnn* where *nnnn* is a 4-digit number. The user should avoid defining symbols with the same form.

4.6.3 Location Counter

The current location within the active program section is accessible via the symbol $. This symbol expands to the address of the currently executing instruction (which is different than the address contained in the program counter (PC) register when executing this instruction). Thus:

```
goto $   ;endless loop
```

will represent code that will jump to itself and form an endless loop. By using this symbol and an offset, a relative jump destination can be specified.

Any address offset added to $ has the native addressability of the target device. So, for baseline and mid-range devices, the offset is the number of instructions away from the current location, as these devices have word-addressable program memory. For PIC18 instructions, which use byte addressable program memory, the offset to

this symbol represents the number of bytes from the current location. As PIC18 instructions must be word aligned, the offset to the location counter should be a multiple of 2. All offsets are rounded down to the nearest multiple of 2.

For example:

```
goto       $+2   ;skip...
movlw      8     ;to here for PIC18 devices, or
movwf      _foo  ;to here for baseline and mid-range devices
```

will skip the `movlw` instruction on baseline or mid-range devices. On PIC18 devices, `goto $+2` will jump to the following instruction; i.e., act like a `nop` instruction.

4.6.4 Accessing SFRs

The symbols for Special Function Registers (SFRs) are not automatically accessible from assembly code. The assembly header file `<xc.inc>` must be included into every module that needs access to these register definitions.

Include the header file using the assembler's `INCLUDE` directive, (see 4.9.31. Include Directive) or use the C preprocessor's `#include` directive. If you are using the latter method, make sure you name the assembly source file using a `.S` extension (upper case), or use the `-xassembler-with-cpp` option when you build (see 3.4.36. X: Specify Source Language Option).

An entire register can be accessed using the pre-defined symbol that has been equated to the register's address by the header. For example, the entire LATA register can be accessed via the symbol `LATA`. This symbol name is also made a preprocessor macro, defined to be a string that is the same as the register's name, allowing you to confirm that the register exists before accessing it. For example:

```
#ifdef LATA  // if this device has a PORTA latch register...
movf   LATA,w
movwf  savedState
#endif
```

Preproccessor macros are also defined for fields (often bits) within each register. These macros expand to be the address of the enclosing register followed by a comma and then the bit offset of that field within the register. This makes the symbol directly usable inside bit-orientated instructions, like `bsf` or `btfsc` etc. For example, bit #5 inside the PORTB register can be accessed using the symbol `RB5`. For example:

```
btfss  RB5
incf  count
```

Some devices have the same SFR field name used in more than one register. In these cases, no symbol will be defined for any of these fields. To access these fields, you must use the SFR name and a pre-defined macro that represent the field's bit position in the register. If required, this method of accessing fields within SFRs can be used with any field in any register. For example, some devices define more than one IPEN bit and so you would need to access IPEN with code similar to:

```
bsf BANKMASK(INTCON0),INTCON0_IPEN_POSN,c     ;set the IPEN bit in the INTCON0
register
```

Symbols are also not defined when the field name is a single letter, for example the carry bit in the STATUS register is called C, but there will be no symbol with this name defined. In some cases, there might be an alternate name provided, for example `CARRY`, but the field can always be access in a manner similar to that shown in the above example.

Note: The manual masking of addresses used as instruction operands (for example using the `BANKMASK()` or `PAGEMASK()` macros or ANDing the operand with a mask) is not necessary when using the `-Wl,--fixupoverflow` option and any of the `ignore`, `warn`, or `lstwarn` arguments.

The `<xc.inc>` header supplies a complete set of field-position equates representing the field's bit position in its register, size (in bits), and a bitmask that can be used to perform bitwise operations on the field. These equates have the form `REGISTERNAME_FIELDNAME_`*`SUFFIX`*, where *`SUFFIX`* can be `POSITION` or its alias `POSN`, `SIZE` or its alias `LENGTH`, or `MASK`. For example, the following are defined for the RA3 field in the

PORTA register:`PORTA_RA3_POSN` and `PORTA_RA3_POSITION`, `PORTA_RA3_SIZE` and `PORTA_RA3_LENGTH`, and `PORTA_RA3_MASK`.

4.6.5 Symbolic Labels

A label is a symbolic alias that is assigned a value equal to the current address within the current psect. They can be used to represent a location in program memory, when they act like the name of a routine that can be called or jumped to, and they can be used to represent a location in data memory, hence act like the name of a variable or object. Labels are not assigned a value until link time.

A label definition consists of any valid assembly identifier that must be followed by a colon, `:`. The definition can appear on a line by itself or it can be positioned to the left of an instruction or assembler directive. Regardless of how they are defined, the assembler list file produced by the assembler will always show labels on a line by themselves.

Here are examples of legitimate labels interspersed with assembly code and used with memory reserved for a variable.

```
PSECT myCode,class=CODE,delta=2
start:
  movlw   1
  goto    fin
more:  clrf _input
  return
PSECT myDATA,class=BANK1,space=1
myVar:
  DS      2       ;2 bytes of storage, please
```

Here, the label `start` will ultimately be assigned the same address as the `movlw` instruction, and `more`, the same address as the `clrf` instruction. The label `myVar` will be assigned same address as the start address of the block of memory reserved by the `DS` directive.

Labels can be used (and are preferred) in assembly code, rather than using an absolute address with other instructions. In this way, they can be used as the target location for jump-type instructions or to load an program or data memory address into a register.

Like C variables, assembly labels have scope. By default, they can be used anywhere in the module in which they are defined. They can be used by code located in the source file before their definition. To make a label accessible in other modules, use the `GLOBAL` directive (see 4.9.29. Global Directive for more information).

The assembler will not output information relating to labels that do not use the `GLOBAL` directive, thus you will not see any such symbols appear in the map or symbol files, for example.

4.7 Expressions and Operators

Expressions can consist of unary operators (one operand, e.g., `not`) or binary operators (two operands, e.g., `+`). The operators allowable in expressions are listed in the table below.

Operators within expressions are evaluated left to right, thus the expression `5 + 1 * 2` will yield the value 12. Use parenthesis to change the order of operator execution, for example `5 + (1 * 2)` will yield the value 7.

With the exception of the equality (e.g. `=`) and relational operators (e.g. `>=`), all the operators listed can be freely combined in both constant and relocatable expressions. Relocatable expressions will be evaluated by the linker at link time, after psects have been positioned and symbol values have been determined.

Table 4-5. Assembly Operators

Operator	Purpose	Example
*	multiplication	`movlw 4*33,w`
+	addition	`bra $+1`
-	subtraction	`DB 5-2`

.....continued

Operator	Purpose	Example
/	division	movlw 100/4
= or eq	equality	IF inp eq 66
> or gt	signed greater than	IF inp > 40
>= or ge	signed greater than or equal to	IF inp ge 66
< or lt	signed less than	IF inp < 40
<= or le	signed less than or equal to	IF inp le 66
<> or ne	signed not equal to	IF inp <> 40
low	low byte of operand	movlw low(inp)
high	high byte of operand	movlw high(1008h)
highword	high 16 bits of operand	DW highword(inp)
mod	modulus	movlw 77mod4
& or and	bitwise AND	clrf inp&0ffh
^	bitwise XOR (exclusive or)	movf inp^80,w
|	bitwise OR	movf inp|1,w
not	bitwise complement	movlw not 055h,w
<< or shl	shift left	DB inp>>8
>> or shr	shift right	movlw inp shr 2,w
rol	rotate left	DB inp rol 1
ror	rotate right	DB inp ror 1
float24	24-bit version of real operand	DW float24(3.3)
nul	tests if macro argument is null	

4.8 Program Sections

Program sections, or psects, are simply a section of code or data. They are a way of grouping together parts of a program (via the psect's name) even though the source code cannot be physically adjacent in the source file, or even where spread over several modules.

A psect is identified by a name and has several attributes. The PSECT assembler directive is used to define a psect. It requires a name argument, which may be followed an comma-separated list of flags which define its attributes. Linker options that can be used to control psect placement in memory are described in 6. Linker. These options can be accessed from the driver using the -Wl driver option (see 3.4.35. Wl: Pass Option To The Linker, Option), negating the need for you to run the linker explicitly.

See Assembler-provided Psects and Linker Classes for a list of all psects that can be supplied by the assembler.

Unless defined as abs (absolute), psects are relocatable.

Code or data that is not explicitly placed into a psect using the PSECT directive will become part of the default (unnamed) psect. As you have no control over where this psect is linked, it recommended that a PSECT directive always be placed before the code and objects.

When writing assembly code, you can use the psects provided once <xc.inc> has been included. These have a similar name and function to the sections used by the 8-bit MPASM assembler.

If you create your own psects, try to associate them with an existing linker class (see 5.3. Default Linker Classes and 6.1.2. C: Associate Linker Class To Psect) otherwise you can need to specify linker options for them to be allocated correctly.

Note, that the length and placement of psects is important. It is easier to write code if all executable code is located in psects that do not cross any device pages boundaries; so, too, if data psects do not cross bank boundaries. The location of psects (where they are linked) must match the assembly code that accesses the psect contents.

4.9 Assembler Directives

Assembler directives, or pseudo-ops, are used in a similar way to instruction mnemonics. With the exception of PAGESEL and BANKSEL, these directives do not generate instructions. The DB, DW and DDW directives place data bytes into the current psect. The directives are listed in the following sections.

Table 4-6. Assembler Directives

Directive	Purpose
ALIGN	Aligns output to the specified boundary.
ASMOPT	Controls whether subsequent code is optimized by the assembler.
BANKISEL	Generates code to select bank of operand for indirect access on some devices.
BANKSEL	Generates code to select bank of operand.
CALLSTACK	Indicates the call stack depth remaining.
COND	Controls inclusion of conditional code in the listing file.
CONFIG	Specifies configuration bits.
DABS	Defines absolute storage.
DB	Defines constant byte(s).
DDW	Defines double-width constant word(s).
DEBUG_SOURCE	Controls debug information.
DLABS	Define linear-memory absolute storage.
DS	Reserves storage.
DW	Defines constant word(s).
ELSE	Alternates conditional assembly.
ELSIF	Alternates conditional assembly.
ENDIF	Ends conditional assembly.
END	Ends assembly.
ENDM	Ends macro definition.
EQU	Defines symbol value.
ERROR	Generates a user-defined error.
EXPAND	Controls expansion of assembler macros in the listing file.
EXTRN	Links with global symbols defined in other modules.
FILE	Indicates the source file that contains the assembly code following.
FNADDR	Indicates a routine's address has been taken.

.....continued

Directive	Purpose
FNARG	Indicates calls in a routine's arguments.
FNBREAK	Breaks links in the call graph.
FNCALL	Indicates call hierarchy.
FNCONF	Indicates call stack settings.
FNINDIR	Indicates indirect calls made by routines.
FNSIZE	Indicates the size of a routines auto and parameter objects.
FNROOT	Indicates the root of a call tree.
GLOBAL	Makes symbols accessible to other modules or allow reference to other global symbols defined in other modules.
IF	Conditional assembly.
INCLUDE	Textually includes the content of the specified file.
IRP	Repeats a block of code with a list.
IRPC	Repeats a block of code with a character list.
LINE	Indicates the line number of the current source file that contains the assembly code following.
LIST	Defines options for listing file.
LOCAL	Defines local tabs.
MACRO	Macro definition.
MESSG	Generates a user-defined advisory message.
NOCOND	Controls inclusion of conditional code in the listing file.
NOEXPAND	Controls expansion of assembler macros in the listing file.
NOLIST	Disable assembly listing.
ORG	Sets location counter within current psect.
PAGELEN	Specifies the length of the listing file page.
PAGESEL	Generates set/clear instruction to set PCLATH bits for this page.
PAGEWIDTH	Specifies the width of the listing file page.
PROCESSOR	Defines the particular chip for which this file is to be assembled.
PSECT	Declares or resumes program section.
PUBLIC	Makes non-EXTRN symbols accessible to other modules or allow reference to other global symbols defined in other modules.
RADIX	Specifies radix for numerical constants.
REPT	Repeats a block of code n times.
SET	Defines or re-defines symbol value.
SIGNAT	Defines function signature.
SUBTITLE	Specifies the subtitle of the program for the listing file.
TITLE	Specifies the title of the program for the listing file.

.....continued	
Directive	**Purpose**
WARN	Generates a user-defined warning.

4.9.1 Align Directive

The ALIGN directive aligns whatever is following, data storage or code etc., to the specified offset boundary within the current psect. The boundary is specified as a number of bytes following the directive.

For example, to align output to a 2-byte (even) address within a psect, the following could be used.

```
ALIGN 2
```

Note that what follows will only begin on an even absolute address if the psect begins on an even address; i.e., alignment is done within the current psect. See 4.9.47.16. Reloc Flag for psect alignment.

The ALIGN directive can also be used to ensure that a psect's length is a multiple of a certain number. For example, if the above ALIGN directive was placed at the end of a psect, the psect would have a length that was always an even number of bytes long.

4.9.2 Asmopt Directive

The ASMOPT *action* directive selectively controls the assembler optimizer when processing assembly code. The allowable actions are shown in Table 4-7.

Table 4-7. Asmopt Actions

Action	Purpose
off	Disables the assembler optimizer for subsequent code.
on	Enables the assembler optimizer for subsequent code.
pop	Retrieves the state of the assembler optimization setting.
push	Saves the state of the assembler optimization setting.

No code is modified after an ASMOPT off directive. Following an ASMOPT on directive, the assembler will perform allowable optimizations.

The ASMOPT push and ASMOPT pop directives allow the state of the assembler optimizer to be saved onto a stack of states and then restored at a later time. They are useful when you need to ensure the optimizers are disabled for a small section of code, but you do not know if the optimizers have previously been disabled.

For example:

```
ASMOPT PUSH  ;store the state of the assembler optimizers
ASMOPT OFF   ;optimizations must be off for this sequence
movlw 0x55
movwf EECON2
movlw 0xAA
movwf EECON2
ASMOPT POP   ;restore state of the optimizers
```

Note that no optimizations are performed by the MPLAB XC8 PIC Assembler and these controls will be ignored.

4.9.3 Bankisel Directive

The BANKISEL directive is used to generate code that ensures the correct data bank will be selected for indirect access of the operand's memory location. The operand should be the symbol or numeric address of an object that resides in data memory, for example

```
PROCESSOR 12F510

#include <xc.inc>
```

```
PSECT udata_bank1
myVar:
  DS  1

PSECT code
setMode:
  movlw myVar
  movwf FSR       ;Load the address of myVar info FSR
  BANKISEL myVar  ;Select the correct bank for myVar
  movlw 055h
  movwf INDF      ;Indirectly write to myVar
```

This directive will set the IRP bit (Mid-range) or STATUS bits (Baseline) appropriately for the symbol argument. For all other devices, this directive will be ignored.

4.9.4 Banksel Directive

The `BANKSEL` directive can be used to generate code to select the data bank of the operand. The operand should be the symbol or address of an object that resides in the data memory (see 4.1.2. Bank And Page Selection).

4.9.5 Callstack Directive

The `CALLSTACK` *`depth`* directive indicates to the assembler the number of call stack levels still available at that particular point in the program.

This directive is used by the assembler optimizers to determine if transformations like procedural abstraction can take place.

Note that no optimizations are performed by the MPLAB XC8 PIC Assembler and this control will be ignored.

4.9.6 Cond Directive

The `COND` directive includes conditional code in the assembly listing file output. See also the `COND` directive in 4.9.38. Nocond.

4.9.7 Config Directive

The `CONFIG` directive allows the configuration bits (or fuses) and id-location registers to be specified in the assembly source file. Configuration bits, or fuses, are used to set up fundamental device operation, such as the oscillator mode, watchdog timer, programming mode and code protection. These bits must be correctly set to ensure your program executes correctly.

The directive has the following forms:

```
CONFIG setting = value
CONFIG register = literal_value
```

Here, *`setting`* is a configuration setting descriptor (e.g., `WDT`) and *`value`* can be either a textual description of the desired state (e.g., `OFF`) or a numerical value. Either the *`setting`* or *`value`* tokens or the *`setting = value`* expression can be surrounded by either double or single quotes to protect them from any macro substitution performed by the preprocessor, for example:

```
CONFIG "WDT = ON"          ;turn on watchdog timer
CONFIG "FEXTOSC" = "ECH"   ;external clock oscillator mode, high power PFM
CONFIG WDTPS = 0x1A        ;specify the timer postscale value
```

Some `value` tokens that appear to be purely numerical are in fact a textual description of the value, for example in the following:

```
config WDTPS = 32
```

the `32` token is a textual description for some devices that indicates a watchdog timer post-scale of 1:32, not the number 32. In such a case, it might not be the number 32 that is programmed into the relevant bits of the configuration register. The assembler will first check if *`value`* represents a predefined string and, only if that is not the case, assume it represents a numerical constant, which will then be subject to the same constraints as other numerical constant operands.

You should never assume that the OFF and ON tokens used in configuration macros equate to 0 and 1, respectively, as that is often not the case.

The *register* field is the name of a configuration or id-location register, and this must always be used with a *value* that is a numerical constant, for example:

```
CONFIG CONFIG1L = 0x8F
```

The available *setting*, *register* and *value* fields are documented in the chipinfo file relevant to your device (i.e. pic_chipinfo.html and pic18_chipinfo.html) and that are located in the docs directory of your compiler installation. Click the link to your target device and the page will show you the settings and values that are appropriate with this pragma. Review your device data sheet for more information.

One CONFIG directive can be used to set each configuration setting; alternatively, several comma-separated configuration settings can be specified by the same directive. The directive can be used as many times as required to fully configure the device.

The following example shows a configuration register being programmed as a whole and programmed using the individual settings contained within that register.

```
; PIC18F67K22
; VREG Sleep Enable bit : Enabled
; LF-INTOSC Low-power Enable bit : LF-INTOSC in High-power mode during Sleep
; SOSC Power Selection and mode Configuration bits : High Power SOSC circuit
selected
; Extended Instruction Set : Enabled
CONFIG RETEN = ON, INTOSCSEL = HIGH, SOSCSEL = HIGH, XINST = ON

; Alternatively
CONFIG CONFIG1L = 0x5D

; IDLOC @ 0x200000
CONFIG IDLOC0 = 0x15
```

All the bits in the Configuration Words should be programmed to prevent erratic program behavior. Do not leave them in their default/unprogrammed state. Not all Configuration bits have a default state of logic high; some have a logic low default state. Consult your device data sheet for more information.

If you are using MPLAB X IDE, take advantage of its built-in tools to generate the required pragmas, so that you can copy and paste them into your source code. See the *MPLAB® X IDE User's Guide* for a description and use of the Configuration Bits window.

4.9.8 Dabs Directive

The DABS directive allows one or more bytes of memory to be reserved at the specified address. The general form of the directive is:

```
DABS memorySpace, address, bytes [,symbol]
```

where *memorySpace* is a number representing the memory space in which the reservation will take place, *address* is the address at which the reservation will take place and *bytes* is the number of bytes that is to be reserved. The symbol is optional and refers to the name of the object at the address.

Specifying a symbol allows you to access the reserved memory in your code. This symbol is automatically made globally accessible and is equated to the address specified in the directive. For example, the symbol, foo, defined by the following directive:

```
DABS 1,0x100,4,foo
```

can be used in code, for example:

```
    movlw 20
    movwf BANKMASK(foo)
```

Note: The manual masking of addresses used as instruction operands (for example using the `BANKMASK()` or `PAGEMASK()` macros or ANDing the operand with a mask) is not necessary when using the `-Wl,--fixupoverflow` option and any of the `ignore`, `warn`, or `lstwarn` arguments.

This directive differs to the `DS` directive in that it can be used to reserve memory at any location, not just within the current psect. Indeed, these directives can be placed anywhere in the assembly code and do not contribute to the currently selected psect in any way. Additionally, objects defined by the `DS` directive inside a psect are allocated free memory by the linker, whereas the allocation address must be specified and manged by the programmer when using the `DABS` directive.

The memory space number is the same as the number specified with the `space` flag option to psects (see 4.9.47.18. Space Flag).

The linker reads this `DABS`-related information from object files and ensures that the reserved addresses are not used for other memory placement.

4.9.9 Db Directive

The `DB` directive is used to initialize bytes of program memory.

The directive takes a comma-separated list of arguments. Each argument can be a numeric value (e.g., `55`), a character constant (e.g. `'T'`), or a string (e.g. `"level"` or `'level'`). Each argument or each character of a string argument is written as one byte into consecutive program memory locations within the current psect.

The encoding of the values written depend on the `delta` flag of the psect that contains the directive. In the following code:

```
PSECT myBytes,class=CODE,delta=2
alabel:
  DB  'X',1,2,3,4,
```

the size of the address unit in the program memory specified by the `delta` psect flag (see 4.9.47.4. Delta Flag) is 2 bytes (true for Baseline and Mid-range PIC devices). In this case, the `DB` directive will initialize each word of program memory with the upper byte set to zero, specifically (in hexadecimal):

```
0058 0001 0002 0003 0004
```

However, on PIC18 devices, which uses program memory psects with the `delta` flag set to 1, no padding will occur. For the code:

```
PSECT myBytes,class=CODE,delta=1
alabel:
  DB  'X',1,2,3,4
```

the following (hexadecimal) data will be defined in the HEX file for the program memory.

```
58 01 02 03 04
```

Strings consist of a sequence of characters enclosed within either single or double quotes. There is no termination character added by the assembler. If you need a nul-terminated string, then specify the termination character as a separate argument to the `DB` directive, for example:

```
PSECT myConst,class=CODE,delta=1
bytes:
        DB "a terminated string",0
```

will define:

```
61 20 74 65 72 6D 69 6E 61 74 65 64 20 73 74 72 69 6E 67 00
```

The `DB` directive cannot be used to created objects in data memory. For that, use the `DS` directive (see 4.9.13. Ds Directive).

4.9.10 DDW Directive

The `DDW` directive is used to initialize 32-bit words of program memory.

The directive takes a comma-separated list of arguments. Each argument can be a numeric value (e.g., `55`), a character constant (e.g. `'T'`), or a string (e.g. `"level"` or `'level'`). Each argument or each character of a string argument is written as four bytes into consecutive program memory locations within the current psect.

In the following code:

```
PSECT myWords,class=CODE,delta=2
alabel:
  DDW  'X',1,2,398,0x8005FFFF
```

the `DDW` directive will initialize each word of program memory with the supplied value, specifically (in hexadecimal):

```
00000058 00000001 00000002 0000018E 8005FFFF
```

Strings consist of a sequence of characters enclosed within either single or double quotes. There is no termination character added by the assembler. If you need a nul-terminated string, then specify the termination character as a separate argument to the `DDW` directive, for example:

```
PSECT myConst,class=CODE,delta=1
words:
        DDW "a terminated string",0
```

will define:

```
00000061 00000020 00000074 00000065 00000072 0000006D 00000069 0000006E 00000061
00000074
00000065 00000064 00000020 00000073 00000074 00000072 00000069 0000006E 00000067
00000000
```

The `DDW` directive cannot be used to created objects in data memory. For that, use the `DS` directive (see 4.9.13. Ds Directive).

4.9.11 Debug Source Directive

The `DEBUG_SOURCE` *`action`* directive controls whether the object-code generated by the assembler should favors debugging assembly or C sources. The allowable actions are shown in Table 4-8.

Table 4-8. Debug Source Actions

Action	Purpose
`C`	Favor debugging C source.
`asm`	Favor debugging assembly source.
`pop`	Retrieves the state of the assembler debug source setting.
`push`	Saves the state of the assembler debug source setting.

This directive controls the generation of information that might affect debugging, for example the debug information associated with labels, which can restrict the full expansion of assembly macros shown in the MPLAB X IDE Disassembly View. The directive only affects how the code is debugged; it does not affect the operation of assembled code.

If this directive is not specified, the `asm` setting is employed. Use this setting for projects built with the PIC Assembler or around assembly code using macros that is part of a C project. The `c` setting will automatically be used in assembly output from the MPLAB XC8 C compiler for assembly generated from, C code.

The `DEBUG_SOURCE push` and `DEBUG_SOURCE pop` directives allow the state of the debug source setting to be saved onto a stack of states and then restored at a later time. They are useful when you need to set a particular

debug source state for a small section of code, but you do not know what state the debug source setting had previously been.

For example:

```
DEBUG_SOURCE push    ;store the state of the debug source setting
DEBUG_SOURCE asm     ;ensure proper debugging of the macro used below
    MY_UNLOCK_MACRO  ;this should expand in the Disassembly View
DEBUG_SOURCE pop     ;restore state of the debug source setting
```

4.9.12 Dlabs Directive

The `DLABS` directive must be used to reserve one or more bytes of memory at the specified linear address on those devices that support linear addressing. Typically, the directive is used to allocate memory for large objects that do not fit within one data bank and which must be accessed via linear addressing. The general form of the directive is:

```
DLABS memorySpace, address, bytes [,symbol]
```

The *`memorySpace`* argument is a number representing the linear memory space. This will be the same number as the banked data space. The *`address`* is the address at which the reservation will take place. This can be specified as either a linear or banked address. The *`bytes`* is the number of bytes that is to be reserved. The symbol is optional and refers to the name of the object at the address.

Specifying a symbol allows you to access the reserved memory in your code. The symbol is automatically made globally accessible and is equated to the address specified in the directive in linear addressing form. For example, the symbol, `foo`, defined by the following directive:

```
DLABS 1,0x120,128,foo
```

will be assigned the linear address 0x20A0 and can be used in code, for example:

```
    movlw low(foo)
    movwf FSR1L
    movlw high(foo)
    movwf FSR1H
```

The memory space number is the same as the number specified with the `space` flag option to psects (see 4.9.47.18. Space Flag).

The linker reads this `DLABS`-related information from object files and ensures that the reserved addresses are not used for other memory placement.

4.9.13 Ds Directive

The `DS` *`units`* directive reserves, but does not initialize, the specified amount of space. The single argument is the number of address units to be reserved. An address unit is determined by the flags used with the psect that holds the directive.

This directive is typically used to reserve bytes for RAM-based objects in the data memory (the enclosing psect's `space` flag set to 1). If the psect in which the directive resides is a bit psect (the psect's `bit` flag was set), the directive reserves the request number of bits. If used in a psect linked into the program memory, it will move the location counter, but not place anything in the HEX file output. Note that on Mid-range and Baseline devices, the size of an address unit in the program memory is 2 bytes (see 4.9.47.4. Delta Flag), so the `DS` pseudo-op will actually reserve words in that instance.

An object is typically defined by using a label and then the `DS` directive to reserve locations at the label location.

Examples:

```
PSECT myVars,space=1,class=BANK2
alabel:
  DS 23    ;reserve 23 bytes of memory
PSECT myBits,space=1,bit,class=COMRAM
```

```
xlabel:
  DS 2+3   ;reserve 5 bits of memory
```

4.9.14 Dw Directive

The `DW` directive is used to initialize 16-bit words of program memory.

The directive takes a comma-separated list of arguments. Each argument can be a numeric value (e.g., `55`), a character constant (e.g. `'T'`), or a string (e.g. `"level"` or `'level'`). Each argument or each character of a string argument is written as two bytes into consecutive program memory locations within the current psect.

In the following code:

```
PSECT myWords,class=CODE,delta=2
alabel:
  DW  'X',1,2,398,5472
```

the `DW` directive will initialize each word of program memory with the supplied value, specifically (in hexadecimal):

```
0058 0001 0002 018E 1560
```

Strings consist of a sequence of characters enclosed within either single or double quotes. There is no termination character added by the assembler. If you need a nul-terminated string, then specify the termination character as a separate argument to the `DW` directive, for example:

```
PSECT myConst,class=CODE,delta=1
words:
        DW "a terminated string",0
```

will define:

```
0061 0020 0074 0065 0072 006D 0069 006E 0061 0074 0065 0064 0020 0073 0074 0072
0069 006E 0067 0000
```

The `DW` directive cannot be used to created objects in data memory. For that, use the `DS` directive (see 4.9.13. Ds Directive).

4.9.15 End Directive

The `END` *label* directive is optional, but if present should be at the very end of the program. It will terminate the assembly and not even blank lines should follow this directive.

If an expression is supplied as an argument, that expression will be used to define the entry point of the program. This is stored in a start record in the object file produced by the assembler. Whether this is of any use will depend on the linker.

For example:

```
END  start_label  ;defines the entry point
```

or

```
END               ;do not define entry point
```

4.9.16 Equ Directive

The `EQU` pseudo-op defines a symbol and equates its value to an expression. For example

```
thomas EQU 123h
```

The identifier `thomas` will be given the value 123h. `EQU` is legal only when the symbol has not previously been defined. See 4.9.51. Set Directive for redefinition of values.

This directive does *not* reserve memory for the symbol specified. Use the DS directive to reserve data memory (see 4.9.13. Ds Directive). An EQU performs a similar function to the preprocessor's #define directive (see 5.1. Preprocessor Directives).

4.9.17 Error Directive

The error directive produces a user-defined build-time error message that will halt the assembler. The message to be printed should this directive be executed is specified as a string. Typically this directive will be made conditional, to detect an invalid situation.

For example:

```
IF MODE
  call process
ELSE
  ERROR "no mode defined"
ENDIF
```

4.9.18 Expand Directive

The EXPAND directive shows code generated by macro expansions in the assembler listing file. See also the NOEXPAND directive in 4.9.39. Noexpand Directive. Assembly macros are discussed in Section 5.2.9.14 "MACRO and ENDM".

4.9.19 Extrn Directive

The EXTRN *identifier* pseudo-op is similar to GLOBAL (see 4.9.29. Global Directive), but can only be used to link in with global symbols defined in other modules. An error will be triggered if you use EXTRN with a symbol that is defined in the same module.

4.9.20 File Directive

The FILE directive indicates the source file that contains the assembly code following. For example

```
FILE "X1_start.s"
```

This directive is used to build debug information associated with a project and it is typically found in intermediate assembly files, placed there by the C code generator. They shouldn't be required for hand-written assembly code.

4.9.21 Fnaddr Directive

The FNADDR *routine* directive tells the linker that a routine has had its address taken, and thus it could be called indirectly. This information is used by the linker when allocating objects to the compiled stack.

4.9.22 Fnarg Directive

The FNARG *routine1*,*routine2* directive tells the linker that the evaluation of the an argument to the *routine1* routine involves a call to *routine2*, thus the argument memories for the two routines on the compiled stack should not overlap.

For example

```
FNARG init,start   ;start is called to obtain an argument for init
```

4.9.23 Fnbreak Directive

The FNBREAK *routine1*,*routine2* directive is used to break links in the call graph information produced by the linker.

It states that any calls to routine1 in trees other than the tree rooted at routine2 should not be considered when checking for routines that appear in multiple call graphs. Memory for routine1's compiled stack objects will only be assigned in the graph for the routine rooted at routine2.

4.9.24 Fncall Directive

The FNCALL *caller*,*callee* directive tells the linker that a routine has been called by another. This information is used by the linker to prevent any stack-based objects they define from overlapping in the compiled stack.

For example

```
FNCALL main,init   ;main calls init
```

4.9.25 Fnconf Directive

The FNCONF *psect*,*autos*,*args* directive is used to supply the linker with configuration information for a call graph.

The first argument is the name of the psect in which the compiled stack should be placed. This is followed by the prefix to be used for auto-type objects and argument-type objects. These prefixes are used with the name of the function that defines the objects.

For example:

```
FNCONF rbss,?a,?
```

tells the linker that the compiled stack should be placed in a psect called rbss, and that any auto variable block start with the string ?a and function argument blocks start with ?. Thus, a routine called foo which defines compiled stack objects using an FNARG directive would create two blocks started by the identifiers ?afoo and ?foo in the psect called rbss.

4.9.26 Fnindir Directive

The FNINDIR *routine*,*signature* directive tells the linker that the named routine has performed an indirect call to a routine which has the indicated signature value. See 4.9.52. Signat Directive for information on how to set signature values. The linker will assume the routine could be calling any other routine that has a matching signature value and which has had its address taken. See 4.9.21. Fnaddr Directive for information on how to indicate that a function has had its address taken.

For example, if a routine called fred performs an indirect call to a function with a signature value of 8249, use the following directive:

```
 FNINDIR _fred,8249
```

4.9.27 Fnroot Directive

The FNROOT *routine* directive tells the linker that *routine* is the root of a call graph. Routines called by this routine are built up using the FNCALL directive, see 4.9.24. Fncall Directive. Each call graph has unique memory assigned to it for the stack-based objects defined by routines within that graph.

4.9.28 Fnsize Directive

The FNSIZE *routine*,*autos*,*args* directive informs the linker of the size of the auto variable and argument area associated with *routine*. These values are used by the linker when building the call graph and assigning addresses to the variable and argument areas.

For example, the directive:

```
FNSIZE fred, 10, 5
```

indicates that the routine fred needs 10 bytes of auto variables and 5 bytes of arguments. See 4.9.25. Fnconf Directive for information how to access these areas of memory.

4.9.29 Global Directive

The GLOBAL *identifier_list* directive declares a list of comma-separated symbols. If the symbols are defined within the current module, they are made public. If the symbols are not defined in the current module, they are made references to public symbols defined in external modules. Thus to use the same symbol in two modules the GLOBAL

directive must be used at least twice: once in the module that defines the symbol to make that symbol public and again in the module that uses the symbol to link in with the external definition.

For example:

```
GLOBAL  lab1,lab2,lab3
```

4.9.30 If, Elsif, Else And Endif Directives

These directives implement conditional assembly.

The argument to `IF` and `ELSIF` should be an absolute expression. If it is non-zero, then the code following it up to the next matching `ELSE`, `ELSIF` or `ENDIF` will be assembled. If the expression is zero, then the code up to the next matching `ELSE` or `ENDIF` will not be output. At an `ELSE`, the sense of the conditional compilation will be inverted, while an `ENDIF` will terminate the conditional assembly block. Conditional assembly blocks can be nested.

These directives do not implement a runtime conditional statement in the same way that the C statement `if` does; they are only evaluated when the code is built. In addition, assembly code in both true and false cases is always scanned and interpreted, but the machine code corresponding to instructions is output only if the condition matches. This implies that assembler directives (e.g., `EQU`) will be processed regardless of the state of the condition expression and should not be used inside an `IF` construct.

For example:

```
IF ABC
    goto aardvark
ELSIF DEF
    goto denver
ELSE
    goto grapes
ENDIF
ENDIF
```

In this example, if `ABC` is non-zero, the first `goto` instruction will be assembled but not the second or third. If `ABC` is zero and `DEF` is non-zero, the second `goto` instruction will be assembled but the first and third will not. If both `ABC` and `DEF` are zero, the third `goto` instruction will be assembled.

4.9.31 Include Directive

The `INCLUDE "`*`filename`*`"` directive causes the specified file to be textually replace this directive. For example:

```
INCLUDE "options.inc"
```

The assembler driver does not pass any search paths to the assembler, so if the include file is not located in the current working directory, the file's full path must be specified with the file name.

Assembly source files with a `.S` extension are preprocessed, thus allowing use of preprocessor directives, such as `#include`, which is an alternative to the `INCLUDE` directive.

4.9.32 Irp And Irpc Directives

The `IRP` and `IRPC` directives operate in a similar way to `REPT`; however, instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list.

In the case of `IRP`, the list is a conventional macro argument list. In the case or `IRPC`, it is each character in one argument. For each repetition, the argument is substituted for one formal parameter.

For example:

```
IRP number,4865h,6C6Ch,6F00h
    DW number
ENDM
```

would expand to:

```
DW 4865h
DW 6C6Ch
DW 6F00h
```

Note that you can use local labels and angle brackets in the same manner as with conventional macros.

The `IRPC` directive is similar, except it substitutes one character at a time from a string of non-space characters.

For example:

```
IRPC char,ABC
    DB 'char'
ENDM
```

will expand to:

```
DB 'A'
DB 'B'
DB 'C'
```

4.9.33 Line Directive

The `LINE` directive indicates the line number of the current source file that contains the assembly code following. For example

```
LINE 26
```

This directive is used to build debug information associated with a project and it is typically found in intermediate assembly files, placed there by the C code generator. They shouldn't be required for hand-written assembly code.

4.9.34 List Directive

The `LIST` directive controls whether listing output is produced.

If the listing was previously turned off using the `NOLIST` directive, the `LIST` directive will turn it back on.

Alternatively, the `LIST` control can include options to control the assembly and the listing. The options are listed in the Table 4-9 table.

Table 4-9. List Directive Options

List Option	Default	Description
`c=`*`nnn`*	80	Set the page (i.e., column) width.
`n=`*`nnn`*	59	Set the page length.
`t=ON|OFF`	OFF	Truncate listing output lines. The default wraps lines.
`p=`*`device`*	n/a	Set the device type.
`x=ON|OFF`	OFF	Turn macro expansion on or off.

4.9.35 Local Directive

The `LOCAL` *`label`* directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the `LOCAL` directive will have a unique assembler generated symbol substituted for them when the macro is expanded. For example:

```
down MACRO count
  LOCAL more
  more: decfsz count
  goto more
ENDM
```

when expanded, will include a unique assembler generated label in place of more. For example:

```
down foobar
```

expands to:

```
??0001 decfsz foobar
goto ??0001
```

If invoked a second time, the label more would expand to `??0002` and multiply defined symbol errors will be averted.

4.9.36 Macro And Endm Directives

The `MACRO … ENDM` directives provide for the definition of assembly macros, optionally with arguments. See 4.9.16. Equ Directive for simple association of a value with an identifier, or 5.1. Preprocessor Directives for the preprocessor's `#define` macro directive, which can also work with arguments.

The `MACRO` directive should be preceded by the macro name and optionally followed by a comma-separated list of formal arguments. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters.

For example:

```
;macro: movlf - Move a literal value into a nominated file register
;args:  arg1 - the literal value to load
;       arg2 - the NAME of the source variable
movlf   MACRO   arg1,arg2
  movlw arg1
  movwf arg2 mod 080h
ENDM
```

When used, this macro will expand to the 2 instructions in the body of the macro, with the formal parameters substituted by the arguments. Thus:

```
movlf 2,tempvar
```

expands to:

```
movlw 2
movwf tempvar mod 080h
```

The `&` character can be used to permit the concatenation of macro arguments with other text, but is removed in the actual expansion. For example:

```
loadPort MACRO port, value
  movlw value
  movwf PORT&port
ENDM
```

will load `PORTA` if `port` is `A` when called, etc. The special meaning of the `&` token in macros implies that you can not use the bitwise `AND` operator, (also represented by `&`), in assembly macros; use the `and` form of this operator instead.

A comment can be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double semicolon, `;;`.

When invoking a macro, the argument list must be comma-separated. If it is desired to include a comma (or other delimiter such as a space) in an argument then angle brackets < and > can be used to quote

If an argument is preceded by a percent sign, `%`, that argument will be evaluated as an expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

The `nul` operator can be used within a macro to test a macro argument, for example:

```
IF nul     arg3  ;argument was not supplied.
...
ELSE             ;argument was supplied
...
ENDIF
```

See 4.9.35. Local Directive for use of unique local labels within macros.

By default, the assembly list file will show macro in an unexpanded format; i.e., as the macro was invoked. Expansion of the macro in the listing file can be shown by using the `EXPAND` assembler directive (see 4.9.18. Expand Directive).

4.9.37 Messg Directive

The `messg` directive produces a user-defined build-time advisory message. Execution of this directive will not prevent the assembler from building. The message to be printed should this directive be executed is specified as a string. Typically this directive will be made conditional, to detect an invalid situation.

For example:

```
IF MODE
  call process
ELSE
  MESSG "no mode defined"
ENDIF
```

4.9.38 Nocond

Using the `NOCOND` directive will prevent conditional code from being included in the assembly list file output. See also the `COND` directive in 4.9.6. Cond Directive.

4.9.39 Noexpand Directive

The `NOEXPAND` directive disables macro expansion in the assembly list file. The macro call will be listed instead. See also the `EXPAND` directive in Section 5.2.10.4 “EXPAND”. Assembly macros are discussed in Section 5.2.9.14 “MACRO and ENDM”.

4.9.40 Nolist Directive

The `NOLIST` directive disables assembler listing output from the location where it appears in the source.

Listing can be re-enabled using the `LIST` directive (see 4.9.34. List Directive).

4.9.41 Org Directive

The `ORG` directive changes the value of the location counter within the current psect. This means that the addresses set with `ORG` are relative to the base address of the psect, which is not determined until link time.

Note: The much-abused `ORG` directive does not move the location counter to the absolute address you specify. Only if the psect in which this directive is placed is absolute and overlaid will the location counter be moved to the specified address. To place objects at a particular address, place them in a psect of their own and link this at the required address using the linkers `-P` option (see 6.1.19. P: Position Psect). The `ORG` directive is not commonly required in programs.

The argument to `ORG` must be either an absolute value, or a value referencing the current psect. In either case, the current location counter is set to the value determined by the argument. It is not possible to move the location counter backward. For example:

```
ORG 100h
```

will move the location counter to the beginning of the current psect plus 100h. The actual location will not be known until link time.

In order to use the ORG directive to set the location counter to an absolute value, the directive must be used from within an absolute, overlaid psect. For example:

```
PSECT absdata,abs,ovrld
ORG 50h
;this is guaranteed to reside at address 50h
```

4.9.42 Page Directive

The PAGE directive causes a new page to be started in the listing output. A Control-L (form feed) character will also cause a new page when it is encountered in the source.

4.9.43 Pagelen Directive

The PAGELEN *nnn* directive sets the length of the assembly listing to be the number of specified lines.

4.9.44 Pagesel Directive

The PAGESEL directive can be used to generate code to select the page of the address operand (see 4.1.2. Bank And Page Selection).

4.9.45 Pagewidth Directive

The PAGEWIDTH *nnn* directive sets the width of the assembly listing to be the number of specified characters.

4.9.46 Processor Directive

The PROCESSOR directive should be used in a module if the assembler source is only applicable to one device. The -mcpu option must always be used when building to specify the target device the code is being built for. If there is a mismatch between the device specified in the directive and in the option, an error will be triggered.

For example:

```
PROCESSOR 18F4520
```

4.9.47 Psect Directive

The PSECT directive declares or resumes a program section.

The directive takes as argument a name and, optionally, a comma-separated list of flags. The allowed flags specify attributes of the psect. They are listed in the Table 4-10 table.

The psect name is in a separate name space to ordinary assembly symbols, so a psect can use the same identifier as an ordinary assembly identifier. However, a psect name cannot be one of the assembler directives, keywords, or psect flags.

Once a psect has been declared, it can be resumed later by another PSECT directive; however, the flags need not be repeated and will be propagated from the earlier declaration. An error is generated if two PSECT directives for the same psect are encountered with contradictory flags, the exceptions being that the reloc, size and limit flags can be respecified without error.

Table 4-10. Psect Flags

Flag	Meaning
abs	psect is absolute.
bit	psect holds bit objects.
class=*name*	Specify class name for psect.
delta=*size*	Size of an addressing unit.
global	psect is global (default).
inline	psect contents (function) can be inlined when called.

.....continued	
Flag	**Meaning**
`keep`	psect will not be deleted after inlining.
`limit=`*`address`*	Upper address limit of psect (PIC18 only).
`local`	psect is unique and will not link with others having the same name.
`lowdata`	psect will be entirely located below the 0x1000 address.
`merge=`*`allow`*	Allow or prevent merging of this psect.
`noexec`	For debugging purposes, this psect contains no executable code.
`note`	psect does not contain any data that should appear in the program image.
`optim=`*`optimizations`*	specify optimizations allowable with this psect.
`ovrld`	psect will overlap same psect in other modules.
`pure`	psect is to be read-only.
`reloc=`*`boundary`*	Start psect on specified boundary.
`size=`*`max`*	Maximum size of psect.
`space=`*`area`*	Represents area in which psect will reside.
`split=`*`allow`*	Allow or prevent splitting of this psect.
`with=`*`psect`*	Place psect in the same page as specified psect.

Some examples of the use of the `PSECT` directive follow:

```
; swap output to the psect called fred
PSECT fred
; swap to the psect bill, which has a maximum size of 100 bytes and which is global
PSECT bill,size=100h,global
; swap to joh, which is an absolute and overlaid psect that is part of the CODE
linker class,
; and whose content has a 2-byte word at each address
PSECT joh,abs,ovrld,class=CODE,delta=2
```

4.9.47.1 Abs Flag

The `abs` psect flag defines the current psect as being absolute; i.e., it is to start at location 0. This does not mean that this module's contribution to the psect will start at 0, since other modules can contribute to the same psect (See also 4.9.47.14. Ovrld Flag).

An `abs`-flagged psect is not relocatable and an error will result if a linker option is issued that attempts to place such a psect at any location.

4.9.47.2 Bit Flag

The `bit` psect flag specifies that a psect holds objects that are 1 bit wide. Such psects will have a scale value of 8, indicating that there are 8 addressable units to each byte of storage and that all addresses associated with this psect will be bit addresses, not byte addresses. Non-unity scale values for psects are indicated in the map file (see 6.3. Map Files).

4.9.47.3 Class Flag

The `class` psect flag specifies a corresponding linker class name for this psect. A class is a range of addresses in which psects can be placed.

Class names are used to allow local psects to be located at link time, since they cannot always be referred to by their own name in a `-P` linker option (as would be the case if there are more than one local psect with the same name).

Class names are also useful where psects need only be positioned anywhere within a range of addresses rather than at a specific address. The association of a class with a psect that you have defined typically means that you do not need to supply a custom linker option to place it in memory.

See 6.1.1. A: Define Linker Class Option for information on how linker classes are defined.

4.9.47.4 Delta Flag

The `delta` psect flag defines the size of the addressable unit. In other words, the number of data bytes that are associated with each address.

With PIC Mid-range and Baseline devices, the program memory space is word addressable; so, psects in this space must use a delta of 2. That is to say, each address in program memory requires 2 bytes of data in the HEX file to define their contents. So, addresses in the HEX file will not match addresses in the program memory.

The data memory space on these devices is byte addressable; so, psects in this space must use a delta of 1. This is the default delta value.

All memory spaces on PIC18 devices are byte addressable; so a delta of 1 (the default) should be used for all psects on these devices.

The redefinition of a psect with conflicting delta values can lead to phase errors being issued by the assembler.

4.9.47.5 Global Flag

The `global` psect flag indicates that the linker should concatenate this psect with global psects in other modules and which have the same name.

Psects are considered global by default, unless the `local` flag is used.

4.9.47.6 Inline Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `inline` psect flag is used by the code generator to tell the assembler that the contents of a psect can be inlined. If this operation is performed, the contents of the `inline` psect will be copied and used to replace calls to the function defined in the psect.

4.9.47.7 Keep Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `keep` psect flag ensures that the psect is not deleted after any inlining by the assembler optimizer. Psects that are candidates for inlining (see 4.9.47.6. Inline Flag) can be deleted after the inlining takes place.

4.9.47.8 Limit Flag

The `limit` psect flag specifies a limit on the highest address to which a psect can extend. If this limit is exceeded when it is positioned in memory, an error will be generated. This is currently only available when building for PIC18 devices.

4.9.47.9 Local Flag

A psect defined using the `local` psect flag will not be combined with other `local` psects from other modules at link time, even if there are others with the same name. Where there are two `local` psects in the one module, they reference the same psect. A `local` psect cannot have the same name as any `global` psect, even one in another module.

Psects which are local and which are not associated with a linker class (see 4.9.47.3. Class Flag) cannot be linked to an address using the `-P` linker option, since there could be more than one psect with this name. Typically these psects define a class flag and they are placed anywhere in that class range.

4.9.47.10 Merge Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `merge` psect flag controls how the psect will be merged with others. This flag can be assigned 0, 1, or not specified. When assigned 0, the psect will never be merged by the assembly optimizer during optimizations. If assigned the value 1, the psect can be merged if other psect attributes allow it and the optimizer can see an advantage in doing so. If this flag is not specified, then merging will not take place.

Typically, merging is only performed on code-based psects (`text` psects).

4.9.47.11 Noexec Flag

The `noexec` psect flag is used to indicate that the psect contains no executable code. This information is only relevant for debugging purposes.

4.9.47.12 Note Flag

The `note` psect flag is used by special psects whose content is intended for assembler or debugger tools, and whose content will not be copied to the final program output. When the `note` flag is specified, several other psect flags are prohibited and their use with the same psect will result in a warning.

4.9.47.13 Optim Flag

The `optim` psect flag is used to indicate the optimizations that can be performed on the psect's content, provided such optimizations are permitted and have been enabled.

This flag has no effect for assembly code built with MPLAB XC8 PIC Assembler, which performs no optimizations.

The optimizations are indicated by a colon-separated list of names, shown in the Table 4-11 table. An empty list implies that no optimizations can be performed on the psect.

Table 4-11. Optim Flag Names

Name	Optimization
`inline`	Allow the psect content to be inlined.
`jump`	Perform jump-based optimizations.
`merge`	Allow the psect's content to be merged with that of other similar psects (PIC10/12/16 devices only).
`pa`	Perform proceedural abstraction.
`peep`	Perform peephole optimizations.
`remove`	Allow the psect to be removed entirely if it is completely inlined.
`split`	Allow the psect to be split into smaller psects if it surpasses size restrictions (PIC10/12/16 devices only).
`empty`	Perform no optimization on this psect.

So, for example, the psect definition:

```
PSECT myText,class=CODE,reloc=2,optim=inline:jump:split
```

allows the assembler optimizer to perform inlining, splitting and jump-type optimizations of the `myText` psect content if those optimizations are enabled. The definition:

```
PSECT myText,class=CODE,reloc=2,optim=
```

disables all optimizations associated with this psect regardless of the optimizer setting.

The `optim` psect flag replaces the use of the separate psect flags: `merge`, `split`, `inline` and `keep`.

4.9.47.14 Ovrld Flag

The `ovrld` psect flag tells the linker that the content of this psect should be overlaid with that from other modules at link time. Normally psects with the same name are concatenated across modules. The contributions to an overlaid psect in the same module are always concatenated.

This flag in combination with the `abs` flag (see 4.9.47.1. Abs Flag) defines a truly absolute psect; i.e., a psect within which any symbols defined are absolute.

4.9.47.15 Pure Flag

The `pure` psect flag instructs the linker that this psect will not be modified at runtime. So, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.

4.9.47.16 Reloc Flag

The `reloc` psect flag allows the specification of a requirement for alignment of the psect on a particular boundary. The boundary specification must be a power of two, for example 2, 8 or 0x40. For example, the flag `reloc=100h` would specify that this psect must start on an address that is a multiple of 0x100 (e.g., 0x100, 0x400, or 0x500).

PIC18 instructions must be word aligned, so a `reloc` value of 2 must be used for any PIC18 psect that contains executable code. All other sections, and all sections for all other devices, can typically use the default `reloc` value of 1.

4.9.47.17 Size Flag

The `size` psect flag allows a maximum size to be specified for the psect, e.g., `size=100h`. This will be checked by the linker after psects have been combined from all modules.

4.9.47.18 Space Flag

The `space` psect flag is used to differentiate areas of memory that have overlapping addresses, but are distinct. Psects that are positioned in program memory and data memory have a different space value to indicate that the program space address 0, for example, is a different location to the data memory address 0.

The memory spaces associated with the space flag numbers are shown in Table 4-12.

Table 4-12. Space Flag Numbers

Space Flag Number	Memory Space
0	Program memory, and EEPROM for PIC18 devices
1	Data memory
2	Reserved
3	EEPROM on Mid-range devices
4	Configuration bit
5	IDLOC
6	Note

Devices that have a banked data space do not use different space values to identify each bank. A full address that includes the bank number is used for objects in this space. So, each location can be uniquely identified. For example, a device with a bank size of 0x80 bytes will use address 0 to 0x7F to represent objects in bank 0, and then addresses 0x80 to 0xFF to represent objects in bank 1, etc.

4.9.47.19 Split Flag

This flag is deprecated. Consider, instead, using the `optim` psect flag.

The `split` psect flag can be assigned 0, 1, or not specified. When assigned 0, the psect will never be split by the assembly optimizer during optimizations. If assigned the value 1, the psect can be split if other psect attributes allow it and the psect is too large to fit in available memory. If this flag is not specified, then the splitability of this psect is based on whether the psect can be merged, see 4.9.47.10. Merge Flag.

4.9.47.20 With Flag

The `with` psect flag allows a psect to be placed in the same page with another psect. For example the flag `with=text` will specify that this psect should be placed in the same page as the `text` psect.

The term `withtotal` refers to the sum of the size of each psect that is placed “with” other psects.

4.9.48 Public Directive

The `PUBLIC` *`identifier_list`* directive declares a list of comma-separated symbols. It acts in the same way as the `GLOBAL` directive (see 4.9.29. Global Directive) but with the additional constraint that any symbols defined cannot also have been specified with the `EXTRN` directive.

For example:

```
PUBLIC  lab1,lab2,lab3
```

4.9.49 Radix Directive

The `RADIX` *radix* directive controls the radix for numerical constants specified in the assembler source files. The allowable radices are shown in Table 4-13.

Table 4-13. Radix Operands

Radix	Meaning
dec	Decimal constants
hex	Hexadecimal constants
oct	Octal constants

4.9.50 Rept Directive

The `REPT` directive temporarily defines an unnamed macro, then expands it a number of times as determined by its argument.

For example:

```
REPT 3
    addwf fred,w
ENDM
```

will expand to:

```
addwf fred,w
addwf fred,w
addwf fred,w
```

(see 4.9.32. Irp And Irpc Directives).

4.9.51 Set Directive

The `SET` directive is equivalent to `EQU` (4.9.16. Equ Directive), except that it allows a symbol to be re-defined without error. For example:

```
thomas SET 0h
```

This directive does *not* reserve memory for the symbol specified. Use the `DS` directive to reserve data memory (see 4.9.13. Ds Directive). A `SET` performs a similar function to the preprocessor's `#define` directive (see 5.1. Preprocessor Directives).

4.9.52 Signat Directive

The `SIGNAT` directive is used to associate a 16-bit signature value with a label. At link time, the linker checks that all signatures defined for a particular label are the same. The linker will produce an error if they are not. The `SIGNAT` directive is used to enforce link time checking of function prototypes and calling conventions.

For example:

```
SIGNAT _fred,8192
```

associates the signature value 8192 with the symbol `_fred`. If a different signature value for `_fred` is present in any object file, the linker will report an error.

Often, this directive is used with assembly language routines that are called from C. The easiest way to determine the signature value used by the MPLAB XC8 C Compiler is to write a C routine with the same prototype as that

required for the assembly routine, and check that function's signature directive argument, as determined by the code generator and as shown in the assembly list file.

4.9.53 Space Directive

The `SPACE` *`nnn`* directive places *`nnn`* blank lines in the assembly listing output.

4.9.54 Subtitle Directive

The `SUBTITLE "`*`string`*`"` directive defines a subtitle to appear at the top of every assembly listing page, but under the title. The subtitle should be enclosed in single or double quotes.

4.9.55 Title Directive

The `TITLE "`*`string`*`"` directive keyword defines a title that will appear at the top of every assembly listing page. The title should be enclosed in single or double quotes.

4.9.56 Warn Directive

The `WARN` directive produces a user-defined build-time warning message. Execution of this directive will not prevent the assembler from building. The warning to be printed should this directive be executed is specified as a string. Typically this directive will be made conditional, to detect an invalid situation.

For example:

```
IF MODE
  movwf modeQ
ELSE
  WARN "MODE is zero - write skipped"
ENDIF
```

5. Assembler Features

The PIC Assembler provided access to a C preprocessor and many predefined psects and identifiers that you can use in your programs.

5.1 Preprocessor Directives

The PIC Assembler accepts several specialized preprocessor directives, in addition to the standard directives. All of these are tabulated below.

Table 5-1. Preprocessor Directives

Directive	Meaning	Example
`#`	Preprocessor null directive, do nothing.	`#`

`#define`	Define preprocessor macro.	`#define SIZE (5)`
		`#define FLAG`
		`#define add(a,b) ((a)+(b))`

`#elif`	Short for `#else #if`.	see `#ifdef`
`#else`	Conditionally include source lines.	see `#if`
`#endif`	Terminate conditional source inclusion.	see `#if`
`#error`	Generate an error message.	`#error Size too big`

`#if`	Include source lines if constant expression true.	`#if SIZE < 10`
		`  c = process(10)`
		`#else`
		`  skip();`
		`#endif`

`#ifdef`	Include source lines if preprocessor symbol defined.	`#ifdef FLAG`
		`  do_loop();`
		`#elif SIZE == 5`
		`  skip_loop();`
		`#endif`

`#ifndef`	Include source lines if preprocessor symbol not defined.	`#ifndef FLAG`
		`  jump();`
		`#endif`

`#include`	Include text file into source.	`#include <stdio.h>` `#include "project.h"`
`#line`	Specify line number and filename for listing	`#line 3 final`
`#`*`nn`* `filename`	(where *`nn`* is a number, and *`filename`* is the name of the source file) the following content originated from the specified file and line number.	`#20 init.c`
`#undef`	Undefines preprocessor symbol.	`#undef FLAG`
`#warning`	Generate a warning message.	`#warning Length not set`

Macro expansion using arguments can use the # character to convert an argument to a string and the ## sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself; for example

```
#define __paste1(a,b)   a##b
#define __paste(a,b)    __paste1(a,b)
```

lets you use the paste macro to concatenate two expressions that themselves can require further expansion. Remember, that once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

5.2 Assembler-provided Psects

The PIC Assembler provides psect definitions (tabulated below), can be used to hold code and data, if required. These psects are available once you include <xc.inc> into your source file and have names that resemble the MPASM directive that creates similar sections in the MPASM assembler. For example, to have instructions placed into the code psect, use the following.

```
#include <xc.inc>
PSECT code
;place instructions here
```

The linker class associated with each of these psects and the device families for which they are defined are indicated in the table. The linker classes shown are also already predefined by the PIC Assembler, so you do not need to define them.

You can instead use psects and linker classes that you define, if required. In situations where a psect must reside at a specific location, you must use a unique psect so that it can be linked independently to others. See 4.9.47. Psect Directive and 6.1.19. P: Position Psect.

Table 5-2. Assembler-provided Psects and Linker Classes

Psect name	Linker class	Target device families	Purpose
code	CODE	All	To hold executable code
edata	EEDATA	All	To hold data in EEPROM
data	STRCODE	Baseline, Mid-range	To hold data in program memory
data	CONST	PIC18	To hold data in program memory
udata	RAM	All	To hold objects allocatable anywhere in GPR
udata_acs	COMRAM	PIC18	To hold objects allocatable in the Access bank GPR
udata_bank*n*	BANKN	All	To hold object allocatable in a particular data memory bank
udata_shr	COMMON	Baseline, Mid-range	To hold objects allocatable in common memory

5.3 Default Linker Classes

The linker uses classes to represent memory ranges in which psects can be linked.

Classes are defined by linker options (see 6.1.1. A: Define Linker Class Option). The assembler driver passes a default set of such options to the linker, based on the selected target device. The names of linker classes are case sensitive.

Psects are typically allocated free memory from the class they are associated with. The association is made using the `class` flag with the `PSECT` directive (see 4.9.47.3. Class Flag). Alternatively, a psect can be explicitly placed into the memory associated with a class using a linker option (see 6.1.19. P: Position Psect).

Classes can represent a single memory range, or multiple ranges. Even if two ranges are contiguous, the address where one range ends and the other begins, forms a boundary, and psects placed in the class can never cross such boundaries. You can create classes that cover the same addresses, but which are divided into different ranges and have different boundaries. This allows you to accommodate psects whose contents makes assumptions about where it or the data it accesses would be located in memory. Memory allocated from one class will also be reserved from other classes that specify the same memory addresses.

To the linker, there is no significance to a class name or the memory it defines.

Memory can be removed from these classes if using the `-mreserve` option (see 3.4.22. Reserve Option), or when subtracting memory ranges using the `-mram` and `-mrom` options (see 3.4.21. Ram Option and 3.4.23. Rom Option).

Other than reserve memory from classes, never change or remove address boundaries specified by a class.

5.3.1 Program Memory Classes

The following linker classes are defined once you include `<xc.inc>` and represent program space memory. Not all classes will be present for each device.

`CODE` Consists of ranges that map to the program memory pages on the target device and are used for psects containing executable code.
On Baseline devices, it can only be used by code that is accessed via a jump table.

`ENTRY` Is relevant for Baseline device psects containing executable code that is accessed via a `call` instruction. Calls can only be to the first half of a page on these devices.
The class is defined in such a way that it spans a full page, but the psects it holds will be positioned so that they start in the first half of the page.
This class is also used in Mid-range devices and will consist of many 0x100 word-long ranges, aligned on a 0x100 boundary.

`STRING` Consists of ranges that are 0x100 words long and aligned on a 0x100 boundary. Thus, it is useful for psects whose contents cannot span a 0x100 word boundary.

`STRCODE` Defines a single memory range that covers the entire program memory. It is useful for psects whose content can appear in any page and can cross page boundaries.

`CONST` Consists of ranges that are 0x100 words long and aligned on a 0x100 boundary. Thus, it is useful for psects whose contents cannot span a 0x100 word boundary.

5.3.2 Data Memory Classes

The following linker classes are defined once you include `<xc.inc>` and represent data space memory. Not all classes will be present for each device.

`RAM` Consists of ranges that cover all the general purpose RAM memory of the target device, but excluding any common (unbanked) memory.
Thus, it is useful for psects that must be placed within any general-purpose RAM bank.

`BIGRAM` Consists of a single memory range that is designed to cover the linear data memory of Enhanced Mid-range devices, or the entire available memory space of PIC18 devices.
It is suitable for any psect whose contents are accessed using linear addressing or which does not need to be contained in a single data bank.

`ABS1` Consists of ranges that cover all the general purpose RAM memory of the target device, including any common (unbanked) memory.
Thus, it is useful for psects that must be placed in general purpose RAM, but can be placed in any bank or the common memory,

BANK*x* (where *x* is a bank number) — each consist of a single range that covers the general purpose RAM in that bank, but excluding any common (unbanked) memory.

COMMON Consists of a single memory range that covers the common (unbanked) RAM, if present, for all Mid-range devices.

COMRAM Consists of a single memory range that covers the common (unbanked) RAM, if present, for all PIC18 devices.

SFR*x* (where *x* is a number) — each consists of a single range that covers the SFR memory in bank x. These classes would not typically be used by programmers as they do not represent general purpose RAM.

5.3.3 Miscellaneous Classes

The following linker classes are defined once you include `<xc.inc>` and represent memory for special purposes. Not all classes will be present for each device.

CONFIG Consists of a single range that covers the memory reserved for configuration bit data. This class would not typically be used by programmers as it does not represent general purpose RAM.

IDLOC Consists of a single range that covers the memory reserved for ID location data in the hex file. This class would not typically be used by programmers as it does not represent general purpose RAM.

EEDATA Consists of a single range that covers the EEPROM memory of the target device, if present. This class is used for psects that contain data that is to be programmed into the EEPROM.

5.4 Linker-Defined Symbols

The linker defines special symbols that can be used to determine where sections that were explicitly linked via an option where located in memory. These symbols can be used in C or assembly code, if required.

The link address of a section can be obtained from the value of a global symbol with name `__L`*name* (two leading underscores) where *name* is the name of the section. For example, `__LbssBANK0` is the low bound of the `bssBANK0` section. The highest address of a section (i.e., the link address plus the size) is represented by the symbol `__H`*name*. If the section has different load and link addresses, the load start address is represented by the symbol `__B`*name*; however these are rarely used with PIC devices.

If a section is implicitly linked via a linker class, that is, it is not placed in memory without the explicit use of a `-P` linker option (see 6.1.19. P: Position Psect), the special symbols associated with it are not assigned an address and will have the value 0. Addresses are only assigned to the symbols when the section is linked using a `-P` linker option. That option can simply place the section anywhere in a linker class, for example `-PmyConstData=CONST`.

Assembly code can use these symbol by globally declaring them (noting the two leading underscore characters in the names), for example:

```
GLOBAL __Lidata
```

5.5 Using a Compiled Stack

Not to be confused with a software stack, which is a dynamic stack allocation accessed via a stack pointer, a compiled stack is a memory area that is designated for the static allocation of local objects that should only consume memory for the duration of the routine with which they are associated. This is similar to the stack-based auto and parameter objects defined by functions in C programs. When using the PIC Assembler, the allocation of memory on the stack is performed by the linker. The stack-based objects requested by each routine are assembled into blocks. Blocks from routines that are not active at the same time can be overlaid in memory, based on information in the program's callgraph, which is constructed from directives added to your program.

A more detailed example of using a compiled stack with the PIC Assembler is provided in the *MPLAB™ XC8 PIC Assembler User's Guide for Embedded Engineers*. A summary of this information is provided here.

The FNCONF directive (see 4.9.25. Fnconf Directive) should be used once per program. It's three arguments indicate to the linker the name of the psect that should be used to hold the compiled stack, the symbol prefix to be used for auto-style objects, and the symbol prefix to be used for parameter objects.

The FNROOT directive (see 4.9.27. Fnroot Directive) should be used once for each routine that forms the root node in a callgraph. The memory allocated to stack objects can be overlapped with that of other routines within the same callgraph, but no overlapping will take place between the stack objects of routines that are in different callgraphs. Typically you will define one callgraph root for the main part of your program and then one for each interrupt routine. This way, the stack memory associated with interrupt routines is kept separate and no data corruption can occur.

The FNSIZE directive (see 4.9.28. Fnsize Directive) should be specified for each routine that needs to have objects plaed on the compiled stack. It takes three arguments, those being the name of a routine, the total number of bytes required for that routine's auto-like objects, and the total number of bytes for its parameter-like objects. The directive can be placed anywhere in your code, but it is often located near the routine it configures.

The FNCALL directive (see 4.9.24. Fncall Directive) is used as many times as required to indicate which routines (that use the compiled stack) are called and from where. From this information, the linker can form the callgraph. As you develop your program, you will need to ensure that there is an FNCALL directive for each unique call that takes place in your code. If a called routine does not define any compiled stack objects, the directive is not required for that routine, but it is good practice to include it anyway, in case there are subsequent changes made to the program.

The linker will create the special symbols to be used for auto-like and parameter objects for all routines that used the FNSIZE directive. The following example of a cut-down program shows the stack being set up, with the main and add routines requesting that they each require space on the stack. The main routines reads the special symbols created by the linker for its auto-like objects; the add routine uses special symbols for its parameters..

```
FNCONF udata_acs,?au_,?pa_          ;setup the stack
FNROOT main                         ;the main routine is a callgraph root

PSECT code
FNSIZE main,4,0                     ;the main routine needs 4 bytes of auto-like
objects
GLOBAL ?au_main                     ;make the symbol created by the linker globally
accessible
main:
    ...
loop:
    movff   ?au_main+0,?pa_add+0    ;load the first byte of the first parameter
    movff   ?au_main+1,?pa_add+1    ;load the second byte of the first parameter
    movff   ?au_main+2,?pa_add+2    ;load the first byte of the second parameter
    movff   ?au_main+3,?pa_add+3    ;load the second byte of the second parameter
FNCALL main,add                     ;the main routine will call the add routine in
the callgraph
    call    add                     ;the actual call
    ...

FNSIZE add,0,4                      ;the add routine needs 4 bytes of parameters
GLOBAL ?pa_add                      ;make the symbol created by the linker globally
accessible
add:
    movf    ?pa_add+2,w,c           ;the add routine uses its parameter symbols
    addwf   ?pa_add+0,f,c
    ...
```

5.6 Assembly List Files

An assembly list file is a human-readable listing, showing the opcodes that are present in the final output and the addresses at which they are located.

The assembler will produce an assembly list file if instructed using the -Wa,-a option. There is an assembly list file produced for each assembly source file.

5.6.1 List File Format

The assembly list files arrange the content into columns, with the general form:

***line* [*address*] [*data*]**

Each ***line*** of the list file has a line number. These numbers relate only to the list file itself and are not associated with the lines numbers in the assembly source file from which the list was generated.

The ***address***, if present, is the address at which any output will appear in the device. This may be a program or data space address, which is determined by the psect in which the line is part of. Look for the first psect directive above the line in question.

The ***data***, if present, represents what is associated with the specified address. If this is an instruction opcode, then the mnemonic for that instruction is shown. Lines that represent lables or content in data memory typically do not show the data field, as there is no assembler output corresponding to those lines. The data can also be a comment, which begins with a semicolon, `;`, or an assembler directive.

The following PIC18 example shows a typical list file. Note the `movlw` instruction, whose opcode is 0E50 at address 746E in program memory and which appears on line 51135 of the list file; the `DS` directive which reserves data space memory at address 100.

```
51131                           PSECT brText,class=CODE,space=0,reloc=2
51132                           ; Clear objects in BANK1
51133                           GLOBAL bank1Data
51134  00746A  EE01  F000           lfsr   0,bank1Data
51135  00746E  0E50                 movlw        80
51136  007470                       clear:
51137  007470  6AEE                     clrf postinc0,c
51138  007472  06E8                     decf wreg
51139  007474  E1FD                     bnz  clear
51140  007476  0012                     return
51141  007452                       PSECT        bank1,class=BANK1,space=1,noexec
51142                           bank1Data:
51143  000100                   input:
51144  000100                      DS      2
```

Provided that the link stage has successfully concluded, the listing file is updated by the linker so that it contains absolute addresses and symbol values. Thus, you can use the assembler list file to determine the position and exact opcodes of instructions. Tick marks "`'`" in the assembly listing, next to addresses or opcodes, indicate that the linker did not update the list file, most likely due to a build error, or a assembler option that stopped compilation before the link stage. For example, in the following listing:

```
85  000A' 027F                    subwf   127,w
86  000B' 1D03                    skipz
87  000C' 2800'                   goto    u15
```

These marks indicate that addresses are just address offsets into their enclosing psect, and that opcodes have not been fixed up. Any address field in the opcode that has not been fixed up is shown with a value of 0.

5.6.2 Psect Information

The assembly list file can be used to determine the name of the psect in which a data object or section of code has been placed.

For global symbols, you can check the symbol table in the map file which lists the psect name with each symbol. For symbols local to a module, find the definition of the symbol in the list file. For labels, it is the symbol's name followed by a colon, '`:`'. Look for the first `PSECT` assembler directive above this code. The name associated with this directive is the psect in which the code is placed (see 4.9.47. Psect Directive).

5.6.3 Symbol Table

At the bottom of each assembly list file is a symbol table. This differs from the symbol table presented in the map file (see 6.3.2.6. Symbol Table) in two ways:

- Only symbols associated with the assembly module, from which the list file is produced (as opposed to the entire program) are listed.
- Local as well as global symbols associated with that module are listed.

Each symbol is listed along with the address it has been assigned.

6. Linker

This chapter describes the operation and the usage of the linker.

The application name of the linker is `hlink`. In most instances it will not be necessary to invoke the linker directly, as the assembler driver, `pic-as`, will automatically execute the linker with all the necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the requirements of the linking process. If psects are not linked correctly, code failure can result.

6.1 Operation

A command to the linker takes the following form:

```
hlink [options] files
```

The *options* are zero or more case-insensitive linker options, each of which modifies the behavior of the linker in some way. The *files* is one or more object files and zero or more library files (`.a` extension).

The options recognized by the linker are listed in the table below and are discussed in the paragraphs that follow.

Table 6-1. Linker Command-line Options

Option	Effect
`-Aclass=low-high ,...`	Specify address ranges for a class.
`-Cpsect=class`	Specify a class name for a global psect.
`-Ctype`	Specify call graph type.
`-Dclass=delta`	Specify a class delta value.
`-Dsymfile`	Produce old-style symbol file.
`-Eerrfile`	Write error messages to errfile.
`-F`	Produce `.o` file with only symbol records.
`-G spec`	Specify calculation for segment selectors.
`-H symfile`	Generate symbol file.
`-H+ symfile`	Generate enhanced symbol file.
`-I`	Ignore undefined symbols.
`-J num`	Set maximum number of errors before aborting.
`-K`	Prevent overlaying function parameter and auto areas.
`-L`	Preserve relocation items in .o file.
`-LM`	Preserve segment relocation items in `.o` file.
`-Mmapfile`	Generate a link map in the named file.
`-N`	Sort symbol table in map file by address order.
`-Nc`	Sort symbol table in map file by class address order.
`-Ns`	Sort symbol table in map file by space address order.
`-Ooutfile`	Specify name of output file.
`-Pspec`	Specify psect addresses and ordering.

.....continued	
Option	**Effect**
-Q*processor*	Specify the device type (for cosmetic reasons only).
-S	Inhibit listing of symbols in symbol file.
-S*class=limit*[,*bound*]	Specify address limit, and start boundary for a class of psects.
-U*symbol*	Pre-enter symbol in table as undefined.
-V*avmap*	Use file avmap to generate an Avocet format symbol file.
-W*warnlev*	Set warning level (-9 to 9).
-W*width*	Set map file width (>=10).
-X	Remove any local symbols from the symbol file.
-Z	Remove trivial local symbols from the symbol file.
--DISL=*list*	Specify disabled messages.
--EDF=*path*	Specify message file location.
--EMAX=*number*	Specify maximum number of errors.
--fixupoverflow=*action*	Specify response when encountering fixup overflows.
--NORLF	Do not relocate list file.
--VER	Print version number and stop.

If the standard input is a file, then this file is assumed to contain the command-line argument. Lines can be broken by leaving a backslash \ at the end of the preceding line. In this fashion, hlink commands of almost unlimited length can be issued. For example, a link command file called x.lnk and containing the following text:

```
-Z -Ox.o -Mx.map \
-Ptext=0,data=0/,bss,nvram=bss/. \
x.o y.o z.o
```

can be passed to the linker by one of the following:

```
hlink @x.lnk
hlink < x.lnk
```

Several linker options require memory addresses or sizes to be specified. The syntax for all of these is similar. By default, the number is interpreted as a decimal value. To force interpretation as a HEX number, a trailing H, or h, should be added. For example, 765FH will be treated as a HEX number.

To build projects you will typically use either the MPLAB XC8 C compiler (xc8-cc) driver or the MPLAB XC8 PIC Assembler (pic-as) driver. These will call the linker for you, passing it a set of default linker options. If you need to modify or supply additional linker options when using a driver, use either the -Wl (see 3.4.35. Wl: Pass Option To The Linker, Option) or -Xlinker (see 3.4.39. Xlinker Option) driver options, which will pass on the suboption to the linker application when it is executed.

6.1.1 A: Define Linker Class Option

The -A*class=low*[-*entry*]-*high*[x*count*] option allows one or more address ranges to be defined as a linker class, which can then be used to locate psects in memory. Addresses should use the h suffix to indicate a hexadecimal value.

Typically, one or more comma-separated ranges are defined by a *low* and *high* address. Ranges do not need to be contiguous. For example:

```
-ACODE=1020h-5FFEh,8000h-BFFEh
```

specifies that the class called CODE represents the two distinct address ranges shown.

Psects can be placed anywhere in these ranges by using the -P option and the class name (see 6.1.19. P: Position Psect), for example:

```
-PmyText=CODE
```

will place the myText psect somewhere in the memory ranges defined by the CODE class. Alternatively, any psect that is associated with the CODE class when it is defined (see 4.9.47.3. Class Flag) will automatically be linked into this range, unless it is explicitly located by another option.

Where multiple same-sized, contiguous address ranges are required, they can be specified with a repeat count following an x character. For example:

```
-ACODE=0h-0FFFFhx16
```

specifies that there are 16 contiguous ranges, each 64k bytes in size, starting from address zero (0x0-FFFF, 0x10000-1FFFF, etc.). Even though the address ranges are contiguous, no psect will straddle a boundary between ranges, or in other words psects linked into the class will be wholly positioned in the range 0x0-FFFF or within the range 0x10000-1FFFF etc. Linking psects in such a class might yield a different placement to the case where they were linked into a class defined by:

```
-ACODE=0h-0FFFFFh
```

for example, which defines the same total address range but which does not include 64k boundaries.

The -A linker option can also specify an entry range, which is typically used for placement of code psects on Baseline devices. For example, the class called ENTRY defined as follows:

```
-AENTRY=00h-0FFh-01FFh,0200h-02FFh-03FEh
```

has two ranges, 0x0-1FF and 0x200-2FF. However, the middle address in the option informs the linker that any psect positioned in these ranges must not start above the second address, that is, any psect linked into this class must start in the entry ranges 0x0-FF or 0x200-2FF. A psect may extend beyond the entry range, provided the start address is located within it. Such classes should be used for Baseline devices, where the reachable destinations of call instructions are restricted. See the *MPLAB® XC8 PIC Assembler User's Guide for Embedded Engineers* for a more thorough example of using such classes.

The -A linker option does not specify the memory space associated with the address range. Once a psect is allocated to a class, the space value of the psect is then assigned to the class (see 4.9.47.18. Space Flag). Nor does this option assign a delta value to a class. If required (typically for Baseline and Mid-range classes used to hold executable code), this must be set with the linker's -D option (see 6.1.4. D: Define Class Delta Value).

6.1.2 C: Associate Linker Class To Psect

The -C*psect=class* option allows a psect to be associated with a specific class. Normally, this is not required on the command line because psect classes are specified in object files (see 4.9.47.3. Class Flag).

6.1.3 C: Specify Call Graph Style

The -C*type* option controls what type of call graph is printed in the map file. The available types are shown in the table.

Table 6-2. Call graph types

Type	Produces
n	No call graph.
c	Only critical paths in the call graph.
s	Standard, short-form call graph (default).
f	Full call graph.

The call graph is generated by the linker, primarily for the purposes of allocating memory to objects in the compiled stack. See the section, Using the Compiled Stack in this guide for more information. Those routines defining stack objects that are not overlaid with other stack objects and that are hence contributing to the program's data memory usage are considered as being on a critical path.

6.1.4 D: Define Class Delta Value

The `-Dclass=delta` option defines the `delta` value for the address ranges specified for a class.

The *`delta`* value should be a number. It represents the number of bytes per addressable unit of objects within the psects. The default value is 1, which is the correct value for most classes needed by PIC devices. The `delta` value assigned to a class should match that assigned to the psects that are linked in the class (see 4.9.47.4. Delta Flag).

PIC Baseline and Mid-range devices have word-accessed program memory, thus the `delta` value associated with any psects linked into this memory should be set to 2. Such a value means that two bytes of data are needed to program a single address location in this space. The data memory on these devices is byte addressable, so a `delta` value of 1 is appropriate for classes representing such memory. The assembler automatically specifies the `-D` linker option for any linker classes it defines and that need a non-unity `delta` value. If you define your own linker classes and those classes will be used to represent program memory addresses, you will need to use this option with each of those classes. For example, the following linker options define a program memory class called `MYCODE` and set a `delta` value of 2 for that class.

```
-AMYCODE=0h-07FFhx4  -DMYCODE=2
```

6.1.5 D: Define Old Style Symbol File

Use the `-Dsymfile` option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the symbol followed by the symbol name.

6.1.6 E: Specify Error File

The `-Eerrfile` option makes the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

6.1.7 F: Produce Symbol-only Object FIle

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes you want to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The `-F` option suppresses data and code bytes from the output file, leaving only the symbol records.

This option can be used when part of one project (i.e., a separate build) is to be shared with another, as might be the case with a bootloader and application. The files for one project are compiled using this linker option to produce a symbol-only object file. That file is then linked with the files for the other project.

6.1.8 G: Use Alternate Segment Selector

When linking programs using segmented, or bank-switched psects, there are two ways the linker can assign segment addresses, or selectors, to each segment. A segment is defined as a contiguous group of psects where each psect in sequence has both its link and load addresses concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker.

By default the segment selector is generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the `reloc=` flag value given to psects at the assembler level (see 4.9.47.16. Reloc Flag). The `-Gspec` option allows an alternate method for calculating the segment selector. The argument to `-G` is a string similar to:

```
A/10h-4h
```

where `A` represents the load address of the segment and `/` represents division. This means “Take the load address of the psect, divide by 10 HEX, then subtract 4.” This form can be modified by substituting `N` for `A`, `*` for `/` (to represent

multiplication) and adding, rather than subtracting, a constant. The token `N` is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

```
N*8+4
```

means “take the segment number, multiply by 8, then add 4.” The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined.

The selector of each psect is shown in the map file (see Section 6.4.2.2 “Psect Information Listed by Module”).

6.1.9 H: Generate Symbol File

The `-H`*`symfile`* option instructs the linker to generate a symbol file. The optional argument *`symfile`* specifies the name of the file to receive the data. The default file name is `l.sym`.

6.1.10 H+: Generate Enhanced Symbol FIle

The `-H+`*`symfile`* option will instruct the linker to generate an enhanced symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *`symfile`* specifies a file to receive the symbol file. The default file name is `l.sym`.

6.1.11 I: Ignore Undefined Symbols

Usually, failure to resolve a reference to an undefined symbol is a fatal error. Using the `-I` option causes undefined symbols to be treated as warnings, instead.

6.1.12 J: Specify Maximum Error Count

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the `-J`*`errcount`* option allows this to be altered.

6.1.13 K: Disable Overlap of Compiled Stack

The `-K` linker option disables the overlap of auto/parameter blocks stored on the compiled stack managed by the linker.

If you are linking C programs, the compiled stack is managed by the compiler's code generator. The linker takes no part in the formation of the compiled stack and this option has no effect.

For hand-written assembly programs built using the PIC Assembler, a compiled stack used to hold auto-like and parameter objects is automatically created once you start using FN-type directives in your code. Such a stack is managed by the linker.

If two routines using the compiled stack are not active at the same time (as indicated by a program's `FNCALL` directives), the blocks of data used by those routines for stack-based objects are automatically overlapped in memory by the linker. The `-K` option prevents this overlap, forcing the linker to assign each block into unique memory. The use of this option might increase the amount of data memory used by a program.

6.1.14 L: Allow Load Relocation

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further “relocation” of the program is done at load time. The `-L` option generates, in the output file, one null relocation record for each relocation record in the input.

6.1.15 LM: Allow Segment Load Relocation

Similar to the `-L` option, the `-LM` option preserves relocation records in the output file, but only segment relocations.

6.1.16 M: Generate Map FIle

The `-M`*`mapfile`* option causes the linker to generate a link map in the named file, or on the standard output, if the file name is omitted. The format of the map file is illustrated in 6.3. Map Files.

6.1.17 N: Specify Symbol Table Sorting

By default the symbol table in the map file is sorted by name. The `-N` option causes it to be sorted numerically, based on the value of the symbol. The `-Ns` and `-Nc` options work similarly except that the symbols are grouped by either their space value, or class.

6.1.18 O: Specify Output Filename

This option allows specification of an output file name for the object file.

6.1.19 P: Position Psect

Psects are linked together and assigned addresses based on information supplied to the linker via `-P`*`spec`* options. The argument to the `-P` option consists of comma-separated sequences with the form:

```
-Ppsect=linkaddr+min/loadaddr+min,psect=linkaddr/loadaddr,...
```

All values can be omitted, in which case a default will apply, depending on previous values. The link address of a psect is the address at which it can be accessed at runtime. The load address is the address at which the psect starts within the output file (HEX or binary file etc.), but it is rarely used by 8-bit PIC devices. The addresses specified can be numerical addresses, the names of other psects, classes, or special tokens.

This argument to this option often contains a comma, so it you are passing this option to the linker from either the MPLAB XC8 C compiler (`xc8-cc`) or the MPLAB XC8 PIC Assembler (`pic-as`) drivers, you might need to use the `-Xlinker` driver option (see 3.4.39. Xlinker Option) rather than the `-Wl` option (see 3.4.35. Wl: Pass Option To The Linker, Option) to avoid the comma in the option argument causing unexpected results.

Examples of the basic and most common forms of this option are:

```
-Ptext10=02000h
```

which places (links) the starting address of psect `text10` at address 0x2000;

```
-PmyData=AUXRAM
```

which places the psect `myData` anywhere in the range of addresses specified by the linker class `AUXRAM` (which would need to be defined using the `-A` option, see 6.1.1. A: Define Linker Class Option), and

```
-PstartCode=0200h,endCode
```

which places `endCode` immediately after the end of `startCode`, which will start at address 0x200.

The additional variants of this option are rarely needed; but, are described below.

If a link or load address cannot be allowed to fall below a minimum value, the `+`*`min`*, suffix indicates the minimum address.

If the link address is a negative number, the psect is linked in reverse order with the top of the psect appearing at the specified address minus one. Psects following a negative address will be placed before the first psect in memory.

If the load address is omitted entirely, it defaults to the link address. If the slash `/` character is supplied with no address following, the load address will concatenate with the load address of the previous psect. For example, after processing the option:

```
-Ptext=0,data=0/,bss
```

the `text` psect will have a link and load address of 0; data will have a link address of 0 and a load address following that of text. The `bss` psect will concatenate with data in terms of both link and load addresses.

A load address specified as a dot character, "`.`" tells the linker to set the load address to be the same as the link address.

The final link and load address of psects are shown in the map file (see 6.3.2.2. Psect Information Listed By Module).

6.1.20 Q: Specify Device

The `-Qprocessor` option allows a device type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the device. There are no behavioral changes attributable to the device type.

6.1.21 S: Omit Symbol Information Form Symbol File

The `-S` option prevents symbol information from being included in the symbol file produced by the linker. Segment information is still included.

6.1.22 S: Place Upper Address Limit On Class

A class of psects can have an upper address limit associated with it. The following example of the `-Sclasslimit[,bound]=` option places a limit on the maximum address of the `CODE` class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code using the psect `limit` flag, (see 4.9.47.8. Limit Flag).

If the *bound* (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example below places the `FARCODE` class of psects at a multiple of 1000h, but with an upper address limit of 6000h.

```
-SFARCODE=6000h,1000h
```

6.1.23 U: Add Undefined Symbol

The `-Usymbol` option will enter the specified symbol into the linker's symbol table as an undefined symbol. This is useful for linking entirely from libraries, or for linking a module from a library where the ordering has been arranged so that by default a later module will be linked.

6.1.24 V: Produce Avocet Symbol File

To produce an Avocet format symbol file, the linker needs to be given a map file using the `-Vavmap` option to allow it to map psect names to Avocet memory identifiers. The *avmap* file will normally be supplied with the compiler, or created automatically by the compiler driver as required.

6.1.25 W: Specify Warning Level/Map Width

The `-Wnum` option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values of *num* >= 10.

`-W9` will suppress all warning messages. `-W0` is the default. Setting the warning level to -9 (`-W-9`) will give the most comprehensive warning messages.

6.1.26 X: Omit Local Symbols From Symbol File

Local symbols can be suppressed from a symbol file with the `-X` option. Global symbols will always appear in the symbol file.

6.1.27 Z: Omit Trivial Symbols From Symbol File

Some local symbols are compiler generated and not of interest in debugging. The `-Z` option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "`klfLSu`". The `-Z` option will strip any local symbols starting with one of these letters, and followed by a digit string.

6.1.28 Disl

The `--disl=messages` option is mainly used by the command-line driver, `pic-as`, to disable particular message numbers. It takes a comma-separate list of message numbers that will be disabled during compilation.

6.1.29 Edf

The `--edf=file` specifies the message description file to use when displaying warning or error messages. The argument to this option should be the full path to the message file. Most applications contain an internal copy of the message file, so this option is not normally required. Use this option if you want to specify an alternate file with updated contents.

A message description file is shipped with the MPLAB XC8 C Compiler and is located in the compiler's `pic/dat` directory. The shipped file is called `buildnumber_en.msgs`. A build date (`yyyymmdd` format) will appear at the beginning of the build number and may help you identify more up to date files.

6.1.30 Emax

The `--emax=number` option is mainly used by the command-line driver, `pic-as`, to specify the maximum number of errors that can be encountered before the assembler terminates. The default number is 10 errors.

This option is applied if compiling using `pic-as`, the command-line driver, and the `-fmax-errors` driver option.

6.1.31 Fixupoverflow Linker Option

The `--fixupoverflow=action` option specifies how the linker will respond should it encounter an overflow when fixing up a symbol.

Fixup is the process of replacing symbolic references in code with an actual value once psects have been located in memory and the address or value of symbols have been resolved. If the linker determines that the value of a symbol is too large to fit into the space reserved for it in the program image, then this is known as a fixup overflow. The default action of the linker when encountering a fixup overflow in hand-written assembly programs built with the `pic-as` driver is to place warning markers in the assembly list file where possible.

The `--fixupoverflow` option customizes the action taken by the linker when a fixup overflow situation is encountered, as described by the `action` arguments in the following table.

Table 6-3. Allowable actions on detection of a fixup overflow

Action	Response
`error`	Trigger an error message
`warn`	Truncate the value to fit and trigger a warning message.
`lstwarn`	Truncate the value to fit and insert a warning marker into the assembly list file where the overflow occurred. (the default response when used with `pic-as`)
`ignore`	Truncate the value to fit and continue with no notification of the fact.

The `warn` and `lstwarn` actions can both be selected with this option, in which case they must be separated by a colon (`:`), for example `--fixupoverflow=warn:lstwarn`.

If an assembly list file has been requested using the `-Wa,-a` assembler option and the `lstwarn` action has been requested, a warning message will be inserted into the list file before the instruction or directive that contains the symbol being fixed up. The message will appear similar to:

```
13   003FF4      6F14                    movwf   _bar,b
14   003FF6
                 warning: (2090) fixup overflow storing 0x202 in 1 byte
                 5102                    movf    _foo,w,b
```

which shows an overflow associated with the symbol `_foo`. In this instance, the symbol has been resolved to the value 0x202, but there are only 7 bits in the `movf` instruction to hold the address offset of the file register being accessed. The linker has truncated the value 0x202 to a 7-bit quantity (0x02) and inserted this into the program image forming the opcode 0x5102.

If warning messages are inserted into a list file, a single advisory message will be issued by the assembler for each affected list file to alert you to this action. If the `lstwarn` action was specified, but an assembly list file was not actually requested, then an alternate warning message will be issued by the assembler.

Fixup overflows in hand-written assembly programs might indicate some sort of error in how the code was written or linked. It is commonly associated with the address of symbols in PIC programs that have not been masked to remove the banking or paging information. In this case, if you do not wish to mask addresses manually (see the Address Masking section in this guide), this option can be used to truncate the addresses and allow the program to build. It should be stressed that this option will, however, suppress the error message associated with any fixup overflow. It is recommended that an action of `warn`, or `lstwarn` be used so that you can monitor which instructions are affected. If there is some other erroneous situation in your program, using this option will in no way address the underlying issue.

6.1.32 Norlf

Use of the `--norlf` option prevents the linker applying fixups to the assembly list file produced by the assembler. This option is normally using by the command line driver, `pic-as`, when performing pre-link stages, but is omitted when performing the final link step so that the list file shows the final absolute addresses.

If you are attempting to resolve fixup errors, this option should be disabled so as to fix up the assembly list file and allow absolute addresses to be calculated for this file. If the assembler driver detects the presence of a preprocessor macro `__DEBUG`, which is equated to 1, then this option will be disabled when building. This macro is set when choosing a Debug build in MPLAB X IDE. So, always have this option selected if you encounter such errors.

6.1.33 Ver

The `--ver` option prints information stating the version and build of the linker. The linker will terminate after processing this option, even if other options and files are present on the command line.

6.2 Psects and Relocation

The linker can read both relocatable object files (`.o` extension) and object-file libraries (`.a` extension). Library files are a collection of object files packaged into a single unit and once unpacked, are processed in the same way as individual object files.

Each object file consists of a number of records. Each record has a type that indicates what sort of information it holds. Some record types hold general information about the target device and its configuration, other records types can hold data; and others, program debugging information.

A lot of the information in object files relates to psects (program sections). Psects are an assembly domain construct and are essentially a block of something, either instructions or data. Everything that contributes to the program is located in a psect. See 4.8. Program Sections for an introductory guide. There is a particular record type that is used to hold the data in psects. The bulk of each object file consists of psect records containing the executable code and some objects.

The linker performs the following tasks.

- Combining the content of all referenced relocatable object files into one.
- Relocation of psects contained in the object files into the available device memory.
- Fixup of symbolic references in content of the psects.

Relocation consists of allocating the psects into the memory of the target device.

The target device memory specification is passed to the linker by the way of linker options. These options are generated by the command-line driver, `pic-as`. There are no linker scripts or means of specifying options in any source file. The default linker options rarely need adjusting. But they can be changed, if required, with caution, using the driver option `-Wl`, (see 3.4.35. Wl: Pass Option To The Linker, Option).

Once the psects have been placed at their final memory locations, symbolic references made within the psect can be replaced with absolute values. This is a process called fixup.

The output of the linker is a single object file. This object file is absolute, since relocation is complete and all code and objects have been assigned an address.

6.2.1 Placing Psects into Memory

All code and objects must be placed in a psect (program section). This groups together similar parts of a program and allows you to link those sections using the psect's name. All psects are allocated memory by the linker, after which, the values for any labels defined in those psects can be determined.

When placing a psect into memory, the linker performs the first of the following operations which matches the situation.

- If the psect specifies the `abs` flag, it is placed at address 0 in the memory space indicated by the psect's `space` flag, or the program memory (default) space if no space has been specified.
- If a `-p` linker option references the psect name, the psect is placed at the location specified by that option in the memory space indicated by the psect's `space` flag, or the program memory (default) space if no space has been specified.
- If the psect is associated with a linker class, the psect is placed at any free location in the address ranges defined by that class.
- If the psect specifies a space number, it is placed at a free location in that memory space (Not recommended).
- The psect is placed in a free location in the program memory (default) space (Not recommended).

Some situations are illegal, for example if you use a `-p` option to place a psect that also uses the `abs` flag, then an error will be issued. It is recommended that psects are always linked using the `abs` flag, using a `-p` option, or via a linker class (the first three of the above methods). If the linker has to position a psect with no guidance from the user (the last two of the above methods), a warning similar to, `(526) psect "wanderer" not specified in -P option (first appears in "not_right.o")`, will be emitted.

In most cases, psects can be linked anywhere in a suitable address range that is dictated by the device. For example, most executable code can be placed anywhere in program memory, or at least anywhere in a program memory page. Data objects can usually be placed anywhere in a data bank. In this case, the easiest way to have these psects linked is to associate them with a linker class. If you are using a psect provided by the PIC Assembler, then these are already associated with a suitable linker class and you do not need to specify any linker options to have them correctly linked. In the following example,

```
PSECT udata_bank1
myVar:
  DS 2
```

the `udata` psect has already been associated with the `RAM` linker class and will be linked anywhere in free memory associated with that class.

If you have created your own psect, you can associate it with any of the existing linker classes provided by the PIC Assembler by using the `class` flag with the psect definition. In the following example, a psect has been created by the programmer to use instead of `udata`.

```
PSECT machData,space=1,class=MDATA
myVar:
  DS 2
```

This psect uses a new class, `MDATA`, which will need to be defined by a linker option. To do that, use, for example, the driver option, `-Wl,-AMDATA=050h-05fh`, which passes the `-A` linker option directly to the linker and which will associate the specified address range with the `MDATA` class.

There are, however, times when a psect must be placed at a specific address. The reset vector code is one good example, as are interrupt routines. In this case, you will need to use a `-p` linker option to place the psect at the desired location. This might be as simple as providing an absolute address, for example using the driver option `-Wl,-pInterrupt=08h` to place the psect called `Interrupt` at address 8, but there are more advanced usages of this option.

Check the *MPLAB® XC8 PIC Assembler User's Guide* for full details concerning the psects and linker classes provided by the PIC Assembler, as well as the linker and driver options mentioned in this section.

6.3 Map Files

The map file contains information relating to the memory allocation of psects and the addresses assigned to symbols within those psects.

6.3.1 Map File Generation

If compilation is being performed via MPLAB X IDE, a map file is generated by default. If you are using the driver from the command line, use the `-Wl,-Map` option to request that the map file be produced (see 3.4.35. Wl: Pass Option To The Linker, Option). Map files are typically assigned the extension `.map`.

Map files are produced by the linker application. If the build is stopped before the linker is executed, then no map file is produced. A map file is produced, even if the linker generates errors and this partially-complete file can help you track down the cause of these errors. However, if the linker did not run to completion, due to too many errors or a fatal error, the map file will not be created. You can use the `-fmax-errors` driver option to increase the number of errors allowed before the linker exits.

6.3.2 Contents

The sections in the map file, in order of appearance, are as follows.

- The assembler name and version number.
- A copy of the command line used to invoke the linker.
- The version number of the object code in the first file linked.
- The machine type.
- A psect summary sorted by the psect's parent object file.
- A psect summary sorted by the psect's CLASS.
- A segment summary.
- Unused address ranges summary.
- The symbol table.
- Information summary for each function.
- Information summary for each module.

Portions of an example map file, along with explanatory text, are shown in the following sections.

6.3.2.1 General Information

At the top of the map file is general information relating to the execution of the linker.

When analyzing a program, always confirm the assembler version number shown at the very top of the map file to ensure you are using the assembler you intended to use.

The device selected with the `-mcpu` option (see 3.4.15. Cpu Option), or the one selected in your IDE, should appear after the **Machine type** entry.

The object code version relates to the file format used by relocatable object files produced by the assembler. Unless either the assembler or linker have been updated independently, this should not be of concern.

A typical map file might begin something like the following cut down example.

```
Linker command line:
--edf=/Applications/Microchip/XC8/2.20/dat/en_msgs.txt -cs -h+main.sym -z \
-Q16F946 -ol.o -Mmain.map -ver=XC8 -ACONST=00h-0FFhx32 \
-ACODE=00h-07FFhx4 -ASTRCODE=00h-01FFFh -AENTRY=00h-0FFhx32 \
-ASTRING=00h-0FFhx32 -ACOMMON=070h-07Fh -ABANK0=020h-06Fh \
-ABANK1=0A0h-0EFh -ABANK2=0120h-016Fh -ABANK3=01A0h-01EFh \
-ARAM=020h-06Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh \
-AABS1=020h-07Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh -ASFR0=00h-01Fh \
-ASFR1=080h-09Fh -ASFR2=0100h-011Fh -ASFR3=0180h-019Fh \
-preset_vec=00h,intentry,init,end_init -ppowerup=CODE -pfunctab=CODE \
-ACONFIG=02007h-02007h -pconfig=CONFIG -DCONFIG=2 -AIDLOC=02000h-02003h \
-pidloc=IDLOC -DIDLOC=2 -AEEDATA=00h-0FFh/02100h -peeprom_data=EEDATA \
-DEEDATA=2 -DCODE=2 -DSTRCODE=2 -DSTRING=2 -DCONST=2 -DENTRY=2 -k \
```

```
startup.o main.o

Object code version is 3.10

Machine type is 16F946
```

The information following **Linker command line:** shows all the command-line options and files that were passed to the linker for the last build. Remember, these are linker options, not command-line driver options.

The linker options are necessarily complex. Fortunately, they rarely need adjusting from their default settings. They are formed by the command-line driver, `pic-as`, based on the selected target device and the specified driver options. You can often confirm that driver options were valid by looking at the linker options in the map file. For example, if you ask the driver to reserve an area of memory, you should see a change in the linker options used.

If the default linker options must be changed, this can be done indirectly through the driver using the driver `-Wl` option (see 3.4.35. Wl: Pass Option To The Linker, Option). If you use this option, always confirm the change appears correctly in the map file.

6.3.2.2 Psect Information Listed By Module

The next section in the map file lists those modules that have made a contribution to the output and information regarding the psects that these modules have defined.

This section is heralded by the line that contains the headings:

```
Name   Link  Load  Length  Selector  Space  Scale
```

Under this on the far left is a list of object files (`.o` extension). Both object files that were generated from source modules and those extracted from object library files (`.a` extension) are shown. In the latter case, the name of the library file is printed before the object file list.

Next to the object file are the psects (under the **Name** column) that were linked into the program from that object file. Useful information about that psect is shown in the columns, as follows.

The linker deals with two kinds of addresses: link and load. Generally speaking, the **Link** address of a psect is the address by which it is accessed at runtime.

The **Load** address, which is often the same as the link address, is the address at which the psect starts within the output file (HEX or binary file etc.). If a psect is used to hold bits, the load address is irrelevant and is used to hold the link address (in bit units) converted into a byte address instead.

The **Length** of the psect is shown in the units that are used by that psect.

The **Selector** is less commonly used and is of no concern when compiling for PIC devices.

The **Space** field is important as it indicates the memory space in which the psect was placed. For Harvard architecture machines, with separate memory spaces (such as the PIC devices), this field must be used in conjunction with the address to specify an exact storage location. A space of 0 indicates the program memory and a space of 1 indicates the data memory (see 4.9.47.18. Space Flag).

The **Scale** of a psect indicates the number of address units per byte. This remains blank if the scale is 1 and shows 8 for psects that hold bit objects. The load address of psects that hold bits is used to display the link address converted into units of bytes, rather than the load address (see 4.9.47.2. Bit Flag).

For example, the following appears in a map file.

```
Name     Link  Load  Length Selector  Space  Scale
ext.o    text    3A    3A      22       30      0
         bss     4B    4B      10       4B      1
         rbit    50     A       2        0      1      8
```

This indicates that one of the files that the linker processed was called `ext.o`.

This object file contained a `text` psect, as well as psects called `bss` and `rbit`.

The psect `text` was linked at address 3A and `bss` at address 4B. At first glance, this seems to be a problem, given that text is 22 words long. However, they are in different memory areas, as indicated by the space flag (0 for `text` and 1 for `bss`), and so they do not even occupy the same memory space.

The psect `rbit` contains bit objects, and this can be confirmed by looking at the scale value, which is 8. Again, at first glance it seems that there could be an issue with `rbit` linked over the top of `bss`. Their space flags are the same, but since `rbit` contains bit objects, its link address is in units of bits. The load address field of `rbit` psect displays the link address converted to byte units, i.e., 50h/8 => Ah.

6.3.2.3 Psect Information Listed By Class

The next section in the map file shows the same psect information but grouped by the psects' class.

This section is heralded by the line that contains the headings:

```
TOTAL   Name  Link  Load  Length
```

Under this are the class names followed by those psects which belong to this class (see 4.9.47.3. Class Flag). These psects are the same as those listed by module in the above section; there is no new information contained in this section, just a different presentation.

6.3.2.4 Segment Listing

The class listing in the map file is followed by a listing of segments. Typically this section of the map file can be ignored by the user.

A segment is a conceptual grouping of contiguous psects in the same memory space, and is used by the linker as an aid in psect placement. There is no segment assembler directive and segments cannot be controlled in any way.

This section is heralded by the line that contains the headings:

```
SEGMENTS   Name  Load  Length  Top  Selector  Space  Class
```

The name of a segment is derived from the psect in the contiguous group with the lowest link address. This can lead to confusion with the psect with the same name. Do not read psect information from this section of the map file.

Again, this section of the map file can be ignored.

6.3.2.5 Unused Address Ranges

The last of the memory summaries show the memory that has *not* been allocated and is still available for use.

This section follows the heading:

```
UNUSED ADDRESS RANGES
```

and is followed by a list of classes and the memory that is still available in each class. If there is more than one memory range available in a class, each range is printed on a separate line. Any paging boundaries located within a class are not displayed. But the column Largest block shows the largest contiguous free space (which takes into account any paging in the memory range). If you are looking to see why psects cannot be placed into memory (e.g., cant-find-space type errors) then this is important information to study.

Note that memory can be part of more than one class, thus the total free space is not simply the addition of all the unused ranges.

6.3.2.6 Symbol Table

The next section in the map file alphabetically lists the global symbols that the program defines. This section has the heading:

```
Symbol Table
```

The symbols listed in this table are:

- Global assembly labels
- Global `EQU`/`SET` assembler directive labels
- Linker-defined symbols

Assembly symbols are made global via the `GLOBAL` assembler directive, see 4.9.29. Global Directive for more information.

Linker-defined symbols act like `EQU` directives. However, they are defined by the linker during the link process, and no definition for them appears in any source or intermediate file (see 5.4. Linker-Defined Symbols).

Each symbol is shown with the psect in which it is defined and the value (usually an address) it has been assigned. There is not any information encoded into a symbol to indicate whether it represents code or data – nor in which memory space it resides.

If the psect of a symbol is shown as (`abs`), this implies that the symbol is not directly associated with a psect. Such is the case for absolute C variables, or any symbols that are defined using an `EQU` directive in assembly.

Note that a symbol table is also shown in each assembler list file. These differ to that shown in the map file as they also list local symbols and they only show symbols defined in the corresponding module.

6.3.2.7 Function Information

Following the symbol table is information relating to each function in the program. This information is identical to the function information displayed in the assembly list file. However, the information from all functions is collated in the one location.

6.3.2.8 Module Information

The final section in the map file shows code usage summaries for each module. Each module in the program will show information similar to the following.

```
Module      Function          Class         Link    Load    Size
main.c
            init              CODE          07D8    0000    1
            main              CODE          07E5    0000    13
            getInput          CODE          07D9    0000    4
main.c estimated size: 18
```

The module name is listed (`main.c` in the above example). The special module name shared is used for data objects allocated to program memory and to code that is not specific to any particular module.

Next, the user-defined and library functions defined by each module are listed along with the class in which that psect is located, the psect's link and load address, and its size (shown as bytes for PIC18 devices and words for other 8-bit devices).

After the function list is an estimated size of the program memory used by that module.

7. Utilities

This chapter discusses some of the utility applications that are bundled with the assembler.

The applications discussed in this chapter are those more commonly used, but you do not typically need to execute them directly. Some of their features are invoked indirectly by the command line driver that is based on the command-line arguments or MPLAB X IDE project property selections.

7.1 Archiver/Librarian

The archiver/librarian program has the function of combining several intermediate files into a single file, known as a library archive file. Library archives are easier to manage and might consume less disk space than the individual files contained in them.

The archiver can build all library archive types needed by the assembler and can detect the format of existing archives.

7.1.1 Using the Archiver/Librarian

The archiver program is called `xc8-ar` and is used to create and edit library archive files. It has the following basic command format:

```
xc8-ar [options] file.a [file.o ...]
```

where *`file`*`.a` represents the library archive being created or edited.

The files following the archive file, if required, are the object (`.o`) modules that are required by the command specified.

The *`options`* is zero or more options, tabulated below, that control the program.

Table 7-1. Archiver Command-line Options

Option	Effect
`-d` *`modules`*	Delete module
`-m` *`modules`*	Re-order modules
`-p`	List modules
`-r` *`modules`*	Replace modules
`-t`	List modules with symbols
`-x` *`modules`*	Extract modules
`--target` *`device`*	Specify target device

When replacing or extracting modules, the names of the modules to be replaced or extracted must be specified. If no names are supplied, all the modules in the archive will be replaced or extracted respectively.

Creating an archive file or adding a file to an existing archive is performed by requesting the archiver to replace the module in the archive. Since the module is not present, it will be appended to the archive.

The archiver creates library archives with the modules in the order in which they were given on the command line. When updating an archive, the order of the modules is preserved. Any modules added to an archive will be appended to the end.

The ordering of the modules in an archive is significant to the linker. If an archive contains a module that references a symbol defined in another module in the same archive, the module defining the symbol should come after the module referencing the symbol.

When using the `-d` option, the specified modules will be deleted from the archive. In this instance, it is an error not to supply any module names.

The `-p` option will list the modules within the archive file.

The `-m` option takes a list of module names and re-orders the matching modules in the archive file so that they have the same order as the one listed on the command line. Modules that are not listed are left in their existing order, and will appear after the re-ordered modules.

7.1.1.1 Examples

Here are some examples of usage of the librarian. The following command:

```
xc8-ar -r myLib.a ctime.o init.o
```

creates a library called `myLib.a` that contains the modules `ctime.o` and `init.o`

The following command deletes the object module `a.o` from the library `lcd.a`:

```
xc8-ar -d lcd.a a.o
```

7.2 Hexmate

The Hexmate application is a post-link-stage utility designed to manipulate Intel HEX files.

Hexmate is automatically invoked by the assembler driver, but it can also be executed as a stand-alone application, if required.

7.2.1 Hexmate Uses

Hexmate can be used for a variety of tasks relating to Intel HEX files. These include the following.

- Merging multiple Intel HEX files into one Intel HEX file.
- Calculating and storing variable-length hash values, such as CRC or SHA.
- Filling unused memory locations with known data sequences.
- Converting INHX32 files to other INHX formats (e.g., INHX8M).
- Detecting specific or partial opcode sequences within a HEX file.
- Finding/replacing specific or partial opcode sequences.
- Providing a map of addresses used in a HEX file.
- Changing or fixing the length of data records in a HEX file.
- Validating checksums within Intel HEX files.

Typical applications for Hexmate might include:

- Merging a bootloader or debug module into a main application at build time.
- Calculating a hash value over a range of program memory and storing its value in program memory or EEPROM.
- Filling unused memory locations with an instruction to send the program counter to a known location if it gets lost.
- Storing a serial number at a fixed address.
- Storing a string (e.g., time stamp) at a fixed address.
- Storing initial values at a particular memory address (e.g., initialize EEPROM).
- Detecting the occurrence of a buggy/restricted instructions.
- Adjusting HEX file to meet the requirements of particular bootloaders.

7.2.2 Hexmate Command-line Options

Run Hexmate directly with the following command format:

```
hexmate [specs,]file1.hex [... [specs,]fileN.hex] [options]
```

where *file1*`.hex` through to *fileN*`.hex` forms a list of input Intel HEX files to merge using Hexmate. The *options* can appear anywhere on the command line and are tabulated below.

If only one HEX file is specified, no merging takes place, but other actions can be performed on the HEX file, as specified by the options.

Hexmate can read and write common 8-, 16-, and 32-bit named formats, which contain only specific subsets of record types. The formats are discussed in 7.2.2.16. Format.

Table 7-2. Hexmate Command-line Options

Option	Effect
`--edf=file`	Specify the message description file.
`--emax=n`	Set the maximum number of permitted errors before terminating.
`--msgdisable=number`	Disable messages with the numbers specified.
`--sla=address`	Set the start linear address for a type 5 record.
`--ssa=address`	Set the start segment address for a type 3 record.
`--ver`	Display version and build information then quit.
`-addressing=units`	Set address fields in all Hexmate options to use word addressing or other.
`-break`	Break continuous data so that a new record begins at a set address.
`-ck=spec`	Calculate and store a hash value.
`-fill=spec`	Program unused locations with a known value.
`-find=spec`	Search and notify if a particular code sequence is detected.
`-find=spec,delete`	Remove the code sequence if it is detected (use with caution).
`-find=spec,replace=spec`	Replace the code sequence with a new code sequence.
`-format=type`	Specify maximum data record length or select INHX variant.
`-help`	Show all options or display help message for specific option.
`-logfile=file`	Save Hexmate analysis of output and various results to a file.
`-mask=spec`	Logically AND a memory range with a bitmask.
`-ofile`	Specify the name of the output file.
`-serial=spec`	Store a serial number or code sequence at a fixed address.
`-size`	Report the number of bytes of data contained in the resultant HEX image.
`-string=spec`	Store an ASCII string at a fixed address.
`-strpack=spec`	Store an ASCII string at a fixed address using string packing.
`-wlevel`	Adjust warning sensitivity.
`+`	Prefix to any option to overwrite other data in its address range, if necessary.

The format or assumed radix of values associated with options are detailed with each option description. Note that any address fields specified in these options are to be entered as HEX file addresses, unless you use the `-addressing` option to change this.

7.2.2.1 Specifications And Filename

Hexmate can process Intel HEX files that use either INHX32 or INHX8M format. Additional specifications can be applied to each HEX file listed on the command line to place restrictions or conditions on how this file should be processed.

If any specifications are used, they must precede the filename. The list of specifications will then be separated from the filename by a comma.

A range restriction can be applied with the specification `rStart-End`, where *`Start`* and *`End`* are both assumed to be hexadecimal values. Hexmate will only process data within the address range restriction. For example:

```
r100-1FF,myfile.hex
```

will use `myfile.hex` as input, but only process data which is addressed within the range 0x100-1FF (inclusive) from that file.

An address shift can be applied with the specification `sOffset`, where *`Offset`* is assumed to be an unqualified hexadecimal value. If an address shift is used, data read from this HEX file will be shifted (by the offset specified) to a new address in the output file. The offset can be either positive or negative. For example:

```
r100-1FFs2000,myfile.hex
```

will shift the block of data from 0x100-1FF in `myfile.hex` to the new address range 0x2100-21FF in the output.

Be careful when shifting sections of executable code. Program code should only be shifted if it is position independent.

7.2.2.2 Override Prefix

When the `+` operator precedes an input file or option, the data obtained from that file or the data generated by that option will take priority and be forced into the output file, overwriting any other data existing at the same addresses.

For example, if `input.hex` contains data at address 0x1000, the option:

```
input.hex +-string@1000="My string"
```

will have the data specified by the `-string` option placed at address 0x1000 in the output file; however:

```
+input.hex -string@1000="My string"
```

will copy the data contained in the input HEX file at address 0x1000 into the final output.

Without this option, Hexmate will issue an error if two sources try to store differing data at the same location.

7.2.2.3 Edf

The `--edf=`*`file`* specifies the message description file to use when displaying warning or error messages. The argument to this option should be the full path to the message file. Most applications contain an internal copy of the message file, so this option is not normally required. Use this option if you want to specify an alternate file with updated contents.

A message description file is shipped with the MPLAB XC8 C Compiler and is located in the compiler's `pic/dat` directory. The shipped file is called *`buildnumber`*`_en.msgs`. A build date (*`yyyymmdd`* format) will appear at the beginning of the build number and may help you identify more up to date files.

7.2.2.4 Emax

The `--emax=`*`num`* option sets the maximum number of errors Hexmate will display before execution is terminated, e.g., `--emax=25`. By default, up to 20 error messages will be displayed.

7.2.2.5 Msgdisable

The `--msgdisable=`*`number`* option allows error, warning or advisory messages to be disabled during execution of Hexmate.

The option is passed a comma-separated list of message numbers that are to be disabled. Any error message numbers in this list are ignored unless they are followed by an `:off` argument. For example:

```
--msgdisable=2031,963
```

If the message list is specified as `0`, then all warnings are disabled.

7.2.2.6 Sla Hexmate Option

The `--sla=address` option allows you to specify the linear start address (SLA) in a type 5 record in an INHX32 or INHX032 output file. For example `--sla=0x10000` will ensure the output HEX file will contain a type 5 record with payload 0x10000, e.g.:

```
:040000050010000F6
```

When this option is used, any input SLA records present in the input files are checked for correct syntax and checksum. An error will be issued if they are not valid. These SLA records will then be discarded with *no* warning message. If the output file format is not INHX32 or INHX032, a warning will be issued and no SLA record will be written; otherwise, one SLA record only will appear in the output, containing the value specified by the option.

If this option is not used, any input SLA records present in the input files are checked for correct syntax and checksum. If there is no discrepancy between the addresses specified by these records, one and only one SLA record with that address is written to the output. If there is a conflict between SLA records present in the input files, a warning message will be emitted and *no* SLA record will appear in the output. If there are no SLA records present in the input files, no SLA record will be written to the output.

7.2.2.7 Ssa Hexmate option

The `--ssa=address` option allows you to specify the segment start address (SSA) in a type 3 record in an INHX16 output file. For example `--ssa=0x10000` will ensure that the output HEX file will contain a type 3 record with payload 0x10000, e.g.:

```
:040000030010000F8
```

When this option is used, any SSA records present in the input files are checked for correct syntax and checksum. An error will be issued if they are not valid. These SSA records will then be discarded with *no* warning message. If the output file format is not INHX16, a warning will be issued and no SSA record will be written; otherwise, one SSA record only will appear in the output, containing the value specified by the option.

If this option is not used, any SSA records present in the input files are checked for correct syntax and checksum. If there are no discrepancy between the addresses specified by these records, one and only one SSA record with that address is written to the output. If there is a conflict between SSA records present in the input files, a warning message will be emitted and *no* SSA record will appear in the output. If there are no SSA records present in the input files, no SSA record will be written to the output.

7.2.2.8 Ver

The `--ver` option will ask Hexmate to print version and build information and then quit.

7.2.2.9 Addressing

The `-addressing=units` option allows the addressing units of any addresses in Hexmate's command line options to be changed from the default value of 1 to a maximum value of 4.

By default, all address arguments specified in Hexmate options are assumed to be byte addresses, as used by Intel HEX files. For example, in the option `-mask=0F@0-FF`, the mask will be performed on any HEX file data from address 0x0 to address 0xFF. In some device architectures, the native addressing format can be something other than byte addressing. For example, a HEX file might contain the bytes 0x0F and 0x55 at addresses 0x200 and 0x201, respectively, but when this HEX file is loaded into a Mid-range PIC device, these bytes will form one word at address 0x100 in the device. In this case, the word value at each device address expands into two byte values at separate addresses in the HEX file. If you prefer to use device addresses with Hexmate options, use this option to specify the mapping between HEX file addresses and device addresses.

This option takes one parameter that indicates the number of HEX file bytes that will be stored in each device address location. The parameter may range from the values 1 thru 4. For 8-bit AVR devices, Baseline, Mid-range, and 24-bit PIC devices, an addressing unit of 2 can be used, if desired, for example, `-addressing=2`. You may then specify device addresses in all Hexmate options. For all other Microchip devices, you would typically use the default addressing unit of 1 byte, for example use `-addressing=1` or omit this option entirely. For these devices, the HEX file and device addresses for any location are the same and no mapping is required.

7.2.2.10 Break

The `-break` option takes a comma-separated list of unqualified hexadecimal addresses. If any of these addresses are encountered in the HEX file, the current data record will conclude and a new data record will recommence from the nominated address.

For example, if the output of Hexmate normally contains:

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:10001000ADDEADDEADDEADDEADDEADDEADDEADDE88
:10002000ADDEADDEADDEADDEADDEADDEADDEADDE78
...
```

then if the `-break=16` option is used, the output will become:

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:06001000ADDEADDEADDE49
:10001600ADDEADDEADDEADDEADDEADDEADDEADDE82
:10002600ADDEADDEADDEADDEADDEADDEADDEADDE72
...
```

Breaking data records can create a distinction between functionally different areas of the program space. Some HEX file readers depend on records being arranged this way.

7.2.2.11 Ck Hexmate Option

The `-ck` option is for calculating a hash value. The usage of this option is:

```
-ck=start-end@dest[+offset][wWidth][tCode[.Base]][gAlgorithm][pPolynomial]
[rRevWidth][sSkipWidth[.SkipBytes]][oXORvalue]
```

where:

- *`start`* and *`end`* specify the hexadecimal address range over which the hash will be calculated. If these addresses are not a multiple of the data width for checksum and Fletcher algorithms, the value zero will be padded into the relevant input word locations that are missing.
- *`dest`* is the hexadecimal address where the hash result will be stored. This address cannot be within the range of addresses over which the hash is calculated.
- *`offset`* is an optional initial hexadecimal value to be used in the hash calculations. It is not used with SHA algorithms.
- *`Width`* is optional and specifies the decimal width of the result. Results can be calculated for byte-widths of 1 to 4 bytes for most algorithms, but it represents the bit width for SHA algorithms. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order. This width argument is not required with any Fletcher algorithm, as they have fixed widths, but it may be used to alter the default endianism of the result.
- *`Code`* is an optional hexadecimal code sequence that will trail each byte in the result. Use this feature if you need each byte of the hash result to be embedded within an instruction or if the hash value has to be padded to allow the device to read it at runtime. For example, `t34` will embed each byte of the result in a `retlw` instruction (bit sequence 0x34xx) on Mid-range PIC devices. If the code sequence specifies multiple bytes, these are stored in big-endian order after the hash bytes, for example `tAABB` will append 0xAA immediately after the hash byte and 0xBB at the following address. The trailing code specification `t0000` will store two 0x00 bytes after each byte of the hash. The code sequence argument can be optionally followed by `.`*`Base`*, where *`Base`* is the number of bytes of hash to be output before the trailing code sequence is appended. A specification of `t11.2`, for example, will output the byte 0x11 after each *two* bytes of the hash result.
- *`Algorithm`* is a decimal integer to select which Hexmate hash algorithm to use to calculate the result. A list of selectable algorithms is provided in the table below. If unspecified, the default algorithm used is 8-bit checksum addition (algorithm 1).
- *`Polynomial`* is a hexadecimal value which is the polynomial to be used if you have selected a CRC algorithm.
- *`RevWidth`* is an optional reverse word width. If this is non-zero, then bytes within each word are read in reverse order when calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 or 2. A zero width disables the reverse-byte feature, as if the `r` suboption was not present. This

suboption is intended for situations when Hexmate is being used to match a CRC produced by a PIC hardware CRC module that uses the Scanner module to stream data to it. This feature will work with all hash types, but has no effect when using any checksum algorithm (algorithms -4 thru 4).

- *SkipWidth* is an optional skip word width. If this is non-zero, then the byte at the highest address within each word is skipped for the purposes of calculating a hash value. Words are aligned to the addresses in the HEX file. At present, the width must be 0 (which disables the skip feature, as if the `s` suboption was not present) or greater than 1. This skip width argument can be optionally followed by `.`*SkipBytes*, where *SkipBytes* is a number representing the number of bytes to skip in each word, for example `s4.2` will skip the two bytes at the highest addresses in each 4-byte word. To avoid processing the 'phantom' 0x00 bytes added to HEX files by the MPLAB XC16 C Compiler in hash calculations, use `s4`.
- *XORvalue* is a hexadecimal value that will be XORed with the hash result before it stored.

The single letter argument tokens are case insensitive, so for example `w2` and `W2` are both valid width arguments to this option.

A typical example of the use of this option to calculate a checksum is:

```
-ck=0-1FFF@2FFE+2100w-2g2
```

This will calculate a checksum (16-bit addition) over the range 0 to 0x1FFF and program the checksum result at address 0x2FFE. The checksum value will be offset by 0x2100. The result will be two bytes wide and stored in little-endian format.

Note that the reverse and skip features act on words that are aligned to the HEX file addresses, not to the starting byte of data in the sequence being processed. In other words, the positions of the words are not affected by the start and end addresses specified in the `-ck` option. Consider this option:

```
-ck=0-5@100w2g5p1021s2
```

which specifies that when calculating the hash value, every second byte be skipped (`s2`) over HEX addresses 0 thru 5. If it is acting on the HEX record (data underlined):

```
:10000000640025000030A75007600770078006400 1C
```

the hash will be calculated from the (hexadecimal) bytes 64, 25, and 03. Processing the same HEX record with an option that uses a different start and end address range (1 thru 6):

```
-ck=1-6@100w2g5p1021s2
```

the hash will be calculated from the (hexadecimal) bytes 25, 03, and 75. These features attempt to mimic data read limitations of code running on the device, and thus the words they use are aligned with device addresses, which are in turn aligned to HEX file addresses.

Table 7-3. Hexmate Hash Algorithm Selection

Selector	Algorithm Description
-5	Reflected cyclic redundancy check (CRC).
-4	Subtraction of 32 bit values from initial value.
-3	Subtraction of 24 bit values from initial value.
-2	Subtraction of 16 bit values from initial value.
-1	Subtraction of 8 bit values from initial value.
1	Addition of 8 bit values from initial value.
2	Addition of 16 bit values from initial value.
3	Addition of 24 bit values from initial value.

.....continued	
Selector	**Algorithm Description**
4	Addition of 32 bit values from initial value.
5	Cyclic redundancy check (CRC).
7	Fletcher's checksum (8 bit calculation, 2-byte result width).
8	Fletcher's checksum (16 bit calculation, 4-byte result width).
10	SHA-2 (currently only SHA256 is supported).

See 7.2.3. Hash Value Calculations for more details about the algorithms that are used to calculate hashes.

7.2.2.12 Fill

The `-fill` option is used for filling unused (unspecified) memory locations in a HEX file with a known value. The usage of this option is:

```
-fill=[wwidth:]fill_expr@address[:end_address]
```

where the arguments have the following meaning:

wwidth signifies the decimal width of each constant in the *fill_expr* and can range from 1 thru 9. If this width is not specified, the default value is two bytes. For example, `-fill=w1:0x55@0:0xF` with fill every unused byte between address 0 and 0xF with the byte value 0x55, for example:

```
:10000000FBEF3FF055555555555555555555555DB
```

whereas `-fill=w2:0x55@0:0xF` with fill every unused byte between the same addresses with the value 0x0055, for example:

```
:10000000FBEF3FF055005500550055005500550D9
```

fill_expr defines the values to fill and consists of *const*, which is a base value to place in the first memory location and optionally with *increment*, which indicates how this base value should change after each use. If the base value specifies more than one byte, these are stored in little-endian byte order. These following show the possible fill expressions:

- *const* fill memory with a repeating constant; i.e., `-fill=0xBEEF@0-0x1FF` fills unused locations starting at address 0 with the values 0xBEEF, 0xBEEF, 0xBEEF, 0xBEEF, etc., for example:

  ```
  :10000000FBEF3FF0EFBEEFBEEFBEEFBEEFBEEFBEC9
  ```

- *const*+=*increment* fill memory with an incrementing constant; i.e., `-fill=0xBEEF+=1@0:0x1FF` attempts to fills with the values 0xBEEF, 0xBEF0, 0xBEF1, 0xBEF2, etc., for example:

  ```
  :10000000FBEF3FF0F1BEF2BEF3BEF4BEF5BEF6BEAE
  ```

  Note that *const* increments with each location scanned, regardless of whether that location is populated or unused.

- *const*-=*increment* fill memory with a decrementing constant; i.e.,`-fill=0xBEEF-=0x10@0:0x1FF` attempts to fills with the values 0xBEEF, 0xBEDF, 0xBECF, 0xBEBF, etc., for example:

  ```
  :10000000FBEF3FF0CFBEBFBEAFBE9FBE8FBE7FBE79
  ```

  Note that *const* decrements with each location scanned, regardless of whether that location is populated or unused.

- *const*,*const*,...,*const* fill memory with a list of repeating constants; i.e., -fill=0xDEAD,0xBEEF@0:0x1FF fills with 0xDEAD, 0xBEEF, 0xDEAD, 0xBEEF, etc., for example:

```
:10000000FBEF3FF0ADDEEFBEADDEEFBEADDEEFBE2F
```

@*address* fills a specific address with *fill_expr*; for example, -fill=0xBEEF@0x1000 puts the byte value 0xEF at addresses 0x1000 when the addressing value is set to 1.

```
:01100000EF00
```

If the -addressing=2 option had been additionally used in the above example, the fill option would place 2-bytes at address 0x2000 and 0x2001.

```
:02200000EFBE31
```

:*end_address* optionally specifies an end address of memory to be filled with *fill_expr*; for example, -fill=0xBEEF@0xF0:0xFF puts 0xBEEF in unused addresses between 0 and 0xFF, inclusive.

```
:1000F000EFBEEFBEEFBEEFBEEFBEEFBEEFBEEFBE98
```

If the address range (multiplied by the -addressing value) is not a multiple of the fill value width, the final location will only use part of the fill value, and a warning will be issued.

The fill values are word-aligned so they start on an address that is a multiple of the fill width. Should the fill value be an instruction opcode, this alignment ensures that the instruction can be executed correctly. Similarly, if the total length of the fill sequence is larger than 1 (and even if the specified width is 1), the fill sequence is aligned to that total length. For example the following fill option, which specifies 2 bytes of fill sequence and a starting address that is not a multiple of 2:

```
-fill=w1:0x11,0x22@0x1100:0x1100c
```

will result in the following HEX record, where the starting address was filled with the second byte of the fill sequence due to this alignment.

```
:0C1001002211221122112211221122112211B1
```

Compare that to when the option is -fill=w1:0x11,0x22@0x1100:0x1100c, which does specify a starting address that is a multiple of 2.

```
:0D10000011221122112211221122112211A0
```

All fill constants (excluding the width specification) can be expressed in (unsigned) binary, octal, decimal or hexadecimal, as per normal C syntax, for example, 1234 is a decimal value, 0xFF00 is hexadecimal and FF00 is illegal.

7.2.2.13 Find

The -find=*opcode* option is used to detect and log occurrences of an opcode or code sequence. The usage of this option is:

```
-find=Findcode[mMask]@Start-End[/Align][w][t"Title"]
```

where:

- *Findcode* is the hexadecimal code sequence to search for. For example, to find a clrf instruction with the opcode 0x01F1, use 01F1 as the sequence. In the HEX file, this will appear as the byte sequence F1 01, that is 0xF1 at HEX address 0 and 0x01 at HEX address 1.
- *Mask* is optional. It specifies a bit mask applied over the *Findcode* value to allow a less restrictive search. It is entered in little endian byte order.
- Start and *End* limit the address range to search.

- *Align* is optional. It specifies that a code sequence can only match if it begins on an address that is a multiple of this value.
- w, if present, will cause Hexmate to issue a warning whenever the code sequence is detected.
- *Title* is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

All numerical arguments are assumed to be hexadecimal values.

Here are some examples.

The option -find=1234@0-7FFF/2w will detect the code sequence 1234h (stored in the HEX file as 34 12) when aligned on a 2 (two) byte address boundary, between 0h and 7FFFh. w indicates that a warning will be issued each time this sequence is found.

In this next example, -find=1234M0F00@0-7FFF/2wt"ADDXY", the option is the same as in last example but the code sequence being matched is masked with 000Fh, so Hexmate will search for any of the opcodes 123*x*h, where *x* is any digit. If a byte-mask is used, it must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by Hexmate will refer to this opcode by the name, ADDXY, as this was the title defined for this search.

When requested, a log file will contain the results of all searches. The -find option accepts whole bytes of HEX data from 1 to 8 bytes in length. Optionally, -find can be used in conjunction with replace or delete (as described separately).

7.2.2.14 Find And Delete

If the delete form of the -find option is used, any matching sequences will be deleted from the output file. This implies removal of the data entirely, not replacing it with zero bytes.

To have the -find option perform deletion, append ,delete to the option, for example:

```
-find=ff@7fe0-7fff,delete
```

This function should be used with extreme caution and is not normally recommended for removal of executable code.

7.2.2.15 Find and Replace

If the replace form of the -find option is used, any matching sequences will be replaced, or partially replaced, with new codes.

To have the -find option perform replacement, append ,replace=*spec* to the option in the following manner.

```
-find=spec,replace=Code[mMask]
```

where:

- *Code* is a hexadecimal code sequence to replace the sequences that match the -find criteria.
- *Mask* is an optional bit mask to specify which bits within *Code* will replace the code sequence that has been matched. This can be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and left unchanged.

For example:

```
-find=ff@7fe0-7fff,replace=55
```

This function should be used with extreme caution.

7.2.2.16 Format

The -format=*type*[,*length*] option can be used to specify a particular named format of INHX file to output and/or to adjust the maximum record length.

The *type*argument specifies a particular named INHX format to generate, as tabulated below. This option might be used to change the format of an input file, for example from an INHX16 to an INHX32 file, but the presence of record types 2, 3, 4, or 5 in the input will prevent conversion to an INHX8 file from any other format. Note that the record types present in a HEX file solely determine the file's format, for example, if only record types 0 and 1 are present in

a HEX file, then that file is considered to have an INHX8M format; if type 4 or 5 records were also present, then it is considered to have INHX32 format; if type 2 or 3 and type 4 or 5 records were present, then the file does not conform to any named format and those records which are not permitted in the specified output format will be removed.

The *length*argument is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16 decimal, with 16 being the default.

The possible types that are supported by this option are listed below. Note that `INHX032` is not an actual named INHX format. Selection of this type generates an INHX32 file, but will also initialize the upper address information to zero. This is a requirement of some device programmers.

Table 7-4. HEX file Formats

Type	Valid record types	Comments
`INHX8M`	0, 1	16-bit wide address field.
`INHX16`	0, 1, 2, 3	20-bit wide address field.
`INHX32`	0,1, 4, 5	32-bit wide address field.
`INHX032`	0,1, 4, 5	INHX32 with initialization of upper address to zero.

7.2.2.17 Help

Using `-help` will list all Hexmate options. Entering another Hexmate option as a parameter of `-help` will show a detailed help message for the given option. For example:

```
-help=string
```

will show additional help for the `-string` Hexmate option.

7.2.2.18 Logfile

The `-logfile` option saves HEX file statistics to the named file. For example:

```
-logfile=output.hxl
```

will analyze the HEX file that Hexmate is generating and save a report to a file named `output.hxl`.

7.2.2.19 Mask

Use the `-mask=`*`spec`* option to logically AND a memory range with a particular bitmask. This is used to ensure that the unimplemented bits in program words (if any) are left blank. The usage of this option is as follows:

```
-mask=hexcode@start-end
```

where *`hexcode`* is a value that will be ANDed with data within the *`start`* to *`end`* address range. All values are assumed to be hexadecimal. Multibyte mask values can be entered in little endian byte order.

7.2.2.20 O: Specify Output File

When using the `-o`*`file`* option, the generated Intel HEX output will be created in the specified file. For example:

```
-oprogram.hex
```

will save the resultant output to `program.hex`. The output file can take the same name as one of its input files but, by doing so, it will replace the input file entirely.

If this option is used without a filename, no output is produced, which may be useful if you want to use Hexmate to only show the size of a HEX file, for example. If this option is not used at all, the content of the output HEX file is printed to the standard output stream.

7.2.2.21 Serial

The `-serial=specs` option will store a particular HEX value sequence at a fixed address. The usage of this option is:

```
-serial=Code[+/-Increment]@Address[+/-Interval][rRepetitions]
```

where:

- *Code* is a hexadecimal sequence to store. The first byte specified is stored at the lowest address.
- *Increment* is optional and allows the value of *Code* to change by this value with each repetition (if requested).
- *Address* is the location to store this code, or the first repetition thereof.
- *Interval* is optional and specifies the address shift per repetition of this code.
- *Repetitions* is optional and specifies the number of times to repeat this code.

All numerical arguments are assumed to be hexadecimal values, except for the *Repetitions* argument, which is decimal value by default.

For example:

```
-serial=000001@EFFE
```

will store HEX code 0x00001 to address 0xEFFE.

Another example:

```
-serial=0000+2@1000+10r5
```

will store 5 codes, beginning with value 0000 at address 0x1000. Subsequent codes will appear at address intervals of +0x10 and the code value will change in increments of +0x2.

7.2.2.22 Size

Using the `-size` option will report the number of bytes of data within the resultant HEX image to standard output. The size will also be recorded in the log file if one has been requested.

7.2.2.23 String

The `-string` option will embed an ASCII string at a fixed address. The usage of this option is:

```
-string@Address[tCode]="Text"
```

where:

- *Address* is assumed to be a hexadecimal value representing the address at which the string will be stored.
- *Code* is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
- *Text* is the string to convert to ASCII and embed.

For example:

```
-string@1000="My favorite string"
```

will store the ASCII data for the string, My favorite string (including the null character terminator), at address 0x1000.

And again:

```
-string@1000t34="My favorite string"
```

will store the same string, trailing every byte in the string with the HEX code 0x34.

7.2.2.24 Strpack

The `-strpack=spec` option performs the same function as `-string`, but with two important differences.

Whereas `-string` stores the full byte corresponding to each character, `-strpack` stores only the lower seven bits from each character. Pairs of 7-bit characters are then concatenated and stored as a 14-bit word rather than in separate bytes. This is known as string packing. This is often useful for Mid-range PIC devices, where the program memory is addressed as 14-bit words. If you intend to use this option, you must ensure that the encoded characters are fully readable and correctly interpreted at runtime.

The second difference between these two options is that the `t` specifier usable with `-string` is not applicable with the `-strpack` option.

7.2.2.25 W: Specify warning level

The `-w`*`level`* option sets a warning level threshold. The *`level`* value can be a digit from -9 thru 9, for example `-w5`.

The warning level determines how pedantic Hexmate is about dubious requests or file content. Each warning has a designated warning level; the higher the warning level, the more important the warning message. If the warning message's warning level exceeds the threshold set with this option, the warning is printed. The default warning level threshold is 0 and will allow all normal warning messages.

7.2.3 Hash Value Calculations

A hash value is a small fixed-size value that is calculated from, and used to represent, all the values in an arbitrary-sized block of data. If that data block is copied, a hash recalculated from the new block can be compared to the original hash. Agreement between the two hashes provides a high level of certainty that the copy is valid. There are many hash algorithms. More complex algorithms provide a more robust verification, but can sometimes be too computationally demanding when used in an embedded environment, particularly for smaller devices.

Hexmate can be used to calculate the hash of a program image that is contained in a HEX file. This hash can be embedded into that same HEX file and burned into the target device along with the program image. At runtime, the target device can run a similar hash algorithm over the program image, now stored in its memory. If the stored and calculated hashes are the same, the embedded program can assume that it has a valid program image to execute.

Hexmate implements several hash algorithms, such as checksums and cyclic redundancy checks, which can be selected to calculate the hash value.

When executing Hexmate explicitly, the `-ck` option requests that a hash be calculated, as described in 7.2.2.11. Ck Hexmate Option.

Some consideration is required when a hash value is being calculated over memory that contains unused memory locations. When executing Hexmate explicitly, consider using the `-fill` option (see 7.2.2.12. Fill) to have these locations programmed with a known value.

Hexmate can produce a hash value from any Intel HEX file, regardless of which compiler produced the file and which device that file is intended to program. However, the architecture of the target device may restrict which memory locations can be read at runtime, thus requiring modification to the way in which Hexmate should perform hash calculations, so that the two hashes are calculated similarly and agree. In addition, some compilers might insert padding or phantom bytes into the HEX file that are not present in the device memory. These bytes might need to be ignored by Hexmate when it calculates a hash value and the following discussion indicates possible solutions.

Not all devices can read the entire width of their program memory. For example, Baseline and Mid-range PIC devices can only read the lower byte of each program memory location. The HEX file, however, will contain two bytes for each program memory word and both these bytes will normally be processed by Hexmate when calculating a hash value. When executing Hexmate explicitly, use the `s2` suboption to the `-ck` option to have the MSB of each 2-byte word skipped. Note, however, that this sort of verification process is not considering corruption in the MSB of each program word.

Some devices have hardware CRC modules which can calculate a CRC hash value. If desired, program memory data can be streamed to this module using the Scanner module to automate the calculation. As the Scanner module reads the MSB of each program memory word first, you need to have Hexmate also process HEX file bytes within an instruction word in the reverse order. When executing Hexmate explicitly, use the `r2` suboption to the `-ck` option to have Hexmate process bytes in a 2-byte word in reverse order.

Some consideration must also be given to how the Hexmate hash value encoded in the HEX file can be read at runtime.

Baseline and Mid-range PIC devices must store data in program memory using `retlw` instructions. Thus they need one instruction to store each byte of the hash value calculated by Hexmate. When executing Hexmate explicitly, use the `t34` suboption to the `-ck` option to have Hexmate process store each bytes as part of a `retlw` instruction.

The dsPIC and PIC24 devices cannot read the full width of their program memory and data to be loaded to this memory is typically stored in 2-byte chunks per every 4 bytes of HEX file data.When executing Hexmate explicitly, use the `t0000.2` suboption to the `-ck` option to have Hexmate store two bytes of the hash value in the lower half of each 4-bytes of the HEX file, with the upper bytes set to zero.

7.2.3.1 Hash Algorithms

The following sections provide examples of the algorithms that Hexmate uses and that can be used to calculate the corresponding hash value at runtime. Note that these examples may require modification for the intended device and situation.

7.2.3.1.1 Addition Algorithms

Hexmate has several simple checksum algorithms that sum data values over a range in the program image. These algorithms correspond to the selector values 1, 2, 3 and 4 in the algorithm suboption and read the data in the program image as 1, 2, 3 or 4 byte quantities, respectively. This summation is added to an initial value (offset) that is supplied to the algorithm via the same option. The width to which the final checksum is truncated is also specified by this option and can be 1, 2, 3 or 4 bytes. Hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below can be customized to work with any combination of data size (`read_t`) and checksum width (`result_t`).

```
#include <stdint.h>
typedef uint8_t read_t;        // size of data values read and summed
typedef uint16_t result_t;     // size of checksum result

// add to offset, n additions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to sum
// n:    the number of sums to perform
// offset: the intial value to which the sum is added
result_t ck_add(const read_t *data, unsigned n, result_t offset)
{
    result_t chksum;
    chksum = offset;
    while(n--) {
        chksum += *data;
        data++;
    }
    return chksum;
}
```

The `read_t` and `result_t` type definitions should be adjusted to suit the data read/sum width and checksum result width, respectively. If you never use an offset, that parameter can be removed and `chksum` assigned 0 before the loop.

Here is how this function might be used when, for example, a 2-byte-wide checksum is to be calculated from the addition of 1-byte-wide values over the address range 0x100 to 0x7fd, starting with an offset of 0x20. The checksum is to be stored at 0x7fe and 0x7ff in little endian format.

When executing Hexmate explicitly, use the option:

```
-ck=100-7fd@7fe+20g1w-2
```

Adapt the following MPLAB XC8 code snippet for PIC devices, which calls `ck_add()` and compares the runtime checksum with that stored by Hexmate at compile time.

```
extern const read_t ck_range[0x6fe/sizeof(read_t)] __at(0x100);
extern const result_t hexmate __at(0x7fe);
result_t result;
```

```
result = ck_add(ck_range, sizeof(ck_range)/sizeof(read_t), 0x20);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

This code uses the placeholder array, `ck_range`, to represent the memory over which the checksum is calculated and the variable `hexmate` is mapped over the locations where Hexmate will have stored its checksum result. Being `extern` and absolute, neither of these objects consume additional device memory. Adjust the addresses and sizes of these objects to match the option you pass to Hexmate.

Hexmate can calculate a checksum over any address range; however, the test function, `ck_add()`, assumes that the start and end address of the range being summed are a multiple of the `read_t` width. This is a non-issue if the size of `read_t` is 1. It is recommended that your checksum specification adheres to this assumption, otherwise you will need to modify the test code to perform partial reads of the starting and/or ending data values. This will significantly increase the code complexity.

7.2.3.1.2 Subtraction Algorithms

Hexmate has several checksum algorithms that subtract data values over a range in the program image. These algorithms correspond to the selector values -1, -2, -3, and -4 in the algorithm suboption and read the data in the program image as 1-, 2-, 3- or 4-byte quantities, respectively. In other respects, these algorithms are identical to the addition algorithms. See 7.2.3.1.1. Addition Algorithms for further information regarding the subtraction algorithms.

The function shown below can be customized to work with any combination of data size (`read_t`) and checksum width (`result_t`).

```
#include <stdint.h>
typedef uint8_t read_t;        // size of data values read and subtracted
typedef uint16_t result_t;     // size of checksum result

// add to offset n subtractions of values starting at address data,
// truncating and returning the result
// data: the address of the first value to subtract
// n:    the number of subtractions to perform
// offset: the intial value to which the subtraction is added
result_t ck_sub(const read_t *data, unsigned n, result_t offset)
{
    result_t chksum;
    chksum = offset;
    while(n--) {
        chksum -= *data;
        data++;
    }
    return chksum;
}
```

Here is how this function might be used when, for example, a 4-byte-wide checksum is to be calculated from the addition of 2-byte-wide values over the address range 0x0 to 0x7fd, starting with an offset of 0x0. The checksum is to be stored at 0x7fe and 0x7ff in little endian format.

When executing Hexmate explicitly, use the option:

```
-ck=0-7fd@7fe+0g-2w-4
```

Adapt the following MPLAB XC8 code snippet for PIC devices, which calls `ck_sub()` and compares the runtime checksum with that stored by Hexmate at compile time.

```
extern const read_t ck_range[0x7fe/sizeof(read_t)] __at(0x0);
extern const result_t hexmate __at(0x7fe);
result_t result;

result = ck_sub(ck_range, sizeof(ck_range)/sizeof(read_t), 0x0);
if(result != hexmate)
    ck_failure();  // take appropriate action
```

7.2.3.1.3 Fletcher Algorithms

Hexmate has several algorithms that implement Fletcher's checksum. These algorithms are more complex, providing a robustness approaching that of a cyclic redundancy check, but with less computational effort. There are two forms of this algorithm which correspond to the selector values 7 and 8 in the algorithm suboption and which implement a 1-byte calculation and 2-byte result, with a 2-byte calculation and 4-byte result, respectively. Hexmate will automatically store the checksum in the HEX file at the address specified in the checksum option.

The function shown below performs a 1-byte-wide addition and produces a 2-byte result.

```
#include <stdint.h>
typedef uint16_t result_t;     // size of fletcher result

result_t
fletcher8(const unsigned char * data, unsigned int n )
{
    result_t sum = 0xff, sumB = 0xff;
    unsigned char tlen;
    while (n) {
        tlen = n > 20 ? 20 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xff) + (sum >> 8);
        sumB = (sumB & 0xff) + (sumB >> 8);
    }
    sum = (sum & 0xff) + (sum >> 8);
    sumB = (sumB & 0xff) + (sumB >> 8);
    return sumB << 8 | sum;
}
```

Here is how this function might be used when, for example, a 2-byte-wide Fletcher hash is to be calculated over the address range 0x100 to 0x7fb, starting with an offset of 0x20. The checksum is to be stored at 0x7fc thru 0x7ff in little endian format.

When executing Hexmate explicitly, use the following option. Note that the width cannot be controlled with the `w` suboption, but the sign of this suboption's argument is used to indicate the required endianism of the result.

```
-ck=100-7bd@7fc+20g8w-4
```

This code can be called in a manner similar to that shown for the addition algorithms (see 7.2.3.1.1. Addition Algorithms).

The code for the 2-byte-addition Fletcher algorithm, producing a 4-byte result is shown below.

```
#include <stdint.h>
typedef uint32_t result_t;     // size of fletcher result

result_t
fletcher16(const unsigned int * data, unsigned n)
{
    result_t sum = 0xffff, sumB = 0xffff;
    unsigned tlen;
    while (n) {
        tlen = n > 359 ? 359 : n;
        n -= tlen;
        do {
            sumB += sum += *data++;
        } while (--tlen);
        sum = (sum & 0xffff) + (sum >> 16);
        sumB = (sumB & 0xffff) + (sumB >> 16);
    }
    sum = (sum & 0xffff) + (sum >> 16);
    sumB = (sumB & 0xffff) + (sumB >> 16);
```

```
    return sumB << 16 | sum;
}
```

7.2.3.1.4 CRC Algorithms

Hexmate has several algorithms that implement the robust cyclic redundancy checks (CRC). There is a choice of two algorithms that correspond to the selector values 5 and -5 in the algorithm suboption and that implement a CRC calculation and reflected CRC calculation, respectively. The reflected algorithm works on the least significant bit of the data first.

The polynomial to be used and the initial value can be specified in the option. Hexmate will automatically store the CRC result in the HEX file at the address specified in the checksum option.

Some devices implement a CRC module in hardware that can be used to calculate a CRC at runtime. These modules can stream data read from program memory using a Scanner module. To ensure that the order of the bytes processed by Hexmate and the CRC/Scanner module are identical, you must specify a reverse word width of 2, which will read each 2-byte word in the HEX file in order, but process the bytes within those words in reverse order. When running Hexmate explicitly, use the `r2` suboption to `-ck`.

The function shown below can be customized to work with any result width (`result_t`). It calculates a CRC hash value using the polynomial specified by the `POLYNOMIAL` macro.

```
#include <stdint.h>
typedef uint16_t result_t;     // size of CRC result
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(result_t))
#define MSb     ((result_t)1 << (WIDTH - 1))

result_t
crc(const unsigned char * data, unsigned n, result_t remainder) {
    unsigned pos;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        remainder ^= ((result_t)data[pos] << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    return remainder;
}
```

The `result_t` type definition should be adjusted to suit the result width.

Here is how this function might be used when, for example, a 2-byte-wide CRC hash value is to be calculated values over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format.

When executing Hexmate explicitly, use the option:

```
-ck=0-ff@100+ffffg5w-2p1021
```

Adapt the following MPLAB XC8 code snippet for PIC devices, which calls `crc()` and compares the runtime hash result with that stored by Hexmate at compile time.

```
extern const unsigned char ck_range[0x100] __at(0x0);
extern const result_t hexmate __at(0x100);
result_t   result;

result = crc(ck_range, sizeof(ck_range), 0xFFFF);
if(result != hexmate){
    // something's not right, take appropriate action
    ck_failure();
```

```
}
// data verifies okay, continue with the program
```

The reflected CRC result can be calculated by reflecting the input data and final result, or by reflecting the polynomial. The functions shown below can be customized to work with any result width (`result_t`). The `crc_reflected_IO()` function calculates a reflected CRC hash value by reflecting the data stream bit positions. Alternatively, the `crc_reflected_poly()` function does not adjust the data stream but reflects instead the polynomial, which in both functions is specified by the `POLYNOMIAL` macro. Both functions use the `reflect()` function to perform bit reflection.

```
#include <stdint.h>
typedef uint16_t result_t;     // size of CRC result
typedef unsigned char read_t;
typedef unsigned int reflectWidth;
// This is the polynomial used by the CRC-16 algorithm we are using.
#define POLYNOMIAL     0x1021
#define WIDTH   (8 * sizeof(result_t))
#define MSb     ((result_t)1 << (WIDTH - 1))
#define LSb     (1)
#define REFLECT_DATA(X)       ((read_t) reflect((X), 8))
#define REFLECT_REMAINDER(X)  (reflect((X), WIDTH))

reflectWidth
reflect(reflectWidth data, unsigned char nBits)
{
    reflectWidth reflection = 0;
    reflectWidth reflectMask = (reflectWidth)1 << nBits - 1;
    unsigned char bitp;
    for (bitp = 0; bitp != nBits; bitp++) {
        if (data & 0x01) {
            reflection |= reflectMask;
        }
        data >>= 1;
        reflectMask >>= 1;
    }
    return reflection;
}

result_t
crc_reflected_IO(const unsigned char * data, unsigned n, result_t remainder) {
    unsigned pos;
    unsigned char reflected;
    unsigned char bitp;
    for (pos = 0; pos != n; pos++) {
        reflected = REFLECT_DATA(data[pos]);
        remainder ^= ((result_t)reflected << (WIDTH - 8));
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & MSb) {
                remainder = (remainder << 1) ^ POLYNOMIAL;
            } else {
                remainder <<= 1;
            }
        }
    }
    remainder = REFLECT_REMAINDER(remainder);
    return remainder;
}

result_t
crc_reflected_poly(const unsigned char * data, unsigned n, result_t remainder) {
    unsigned pos;
    unsigned char bitp;
    result_t rpoly;
    rpoly = reflect(POLYNOMIAL, WIDTH);
    for (pos = 0; pos != n; pos++) {
        remainder ^= data[pos];
```

```
        for (bitp = 8; bitp > 0; bitp--) {
            if (remainder & LSb) {
                remainder = (remainder >> 1) ^ rpoly;
            } else {
                remainder >>= 1;
            }
        }
    }
    return remainder;
}
```

Here is how this function might be used when, for example, a 2-byte-wide reflected CRC result is to be calculated over the address range 0x0 to 0xFF, starting with an initial value of 0xFFFF. The result is to be stored at 0x100 and 0x101 in little endian format.

When executing Hexmate explicitly, instead use the following option, noting that the algorithm selected is negative 5 in this case.

```
-ck=0-ff@100+ffffg-5w-2p1021
```

In your project, call either the `crc_reflected_IO()` or `crc_reflected_poly()` functions, as shown previously.

7.2.3.1.5 SHA Algorithms

Hexmate implements a secure hash algorithm (SHA). The selector value 10 selects the SHA256 algorithm, a 256-bit variant of the SHA-2 algorithm.

The code to implement a SHA256 is more complex than other algorithms supported by Hexmate, and as its name suggests, the result of such a hash is 256 bits (32 bytes) wide. Public-domain implementations of this algorithm are available for download from third-party websites, such as github.com/B-Con/crypto-algorithms.

Here is how Hexmate might be used when, for example, a SHA256 hash value is to be calculated values over the address range 0x0 to 0x1FF. The result is to be stored at a starting address of 0x1000 in little endian format.

```
-ck=0-1ff@1000g10w-256
```

8. Error and Warning Messages

Listed here are the MPLAB XC8 PIC Assembler error, warning, and advisory messages, with an explanation of each message. This is the complete and historical message set covering all former HI-TECH C compilers and all compiler versions. Not all messages shown here will be relevant for the compiler version you are using.

Messages have been assigned a unique number that appears in brackets before each message description. It is also printed by the compiler when the message is issued. The messages shown here are sorted by their number.

The name of the application(s) that could have produced the messages are listed in brackets opposite the error message. In some cases examples of code or options that could trigger the error are given. The use of * in the error message is used to represent a string that the compiler will substitute that is specific to that particular error.

Note that one problem in your C or assembler source code can trigger more than one error message. You should attempt to resolve errors or warnings in the order in which they are displayed.

8.1 Messages 0 Thru 499

(1) too many errors (*) (all applications)
The executing compiler application has encountered too many errors and will exit immediately. Other uncompiled source files will be processed, but the compiler applications that would normally be executed in due course will not be run. The number of errors that can be accepted is controlled using the `-fmax-errors` option (see Max Errors Option).

(2) error/warning (*) generated but no description available (all applications)
The executing compiler application has emitted a message (advisory/warning/error), but there is no description available in the message description file (MDF) to print. This could be because the MDF is out-of-date, or the message issue has not been translated into the selected language.

(3) malformed error information on line * in file * (all applications)
The compiler has attempted to load the messages for the selected language, but the message description file (MDF) was corrupted and could not be read correctly.

(100) unterminated #if[n][def] block from line * (Preprocessor)
A `#if` or similar block was not terminated with a matching `#endif`, for example:

```
#if INPUT        /* error flagged here */
int main(void)
{
    run();
}                /* no #endif was found in this module */
```

(101) #* cannot follow #else (Preprocessor)
A `#else` or `#elif` has been used in the same conditional block as a `#else`. These can only follow a `#if`, for example:

```
#ifdef FOO
    result = foo;
#else
    result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
    result = next(0);
#endif
```

(102) #* must be in an #if (Preprocessor)
The `#elif`, `#else` or `#endif` directive must be preceded by a matching `#if` line. If there is an apparently corresponding `#if` line, check for things like extra `#endif`s, or improperly terminated comments, for example:

```
#ifdef FOO
    result = foo;
#endif
    result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
    result = next(0);
#endif
```

(103) #error: * (Preprocessor)
This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines, etc. Remove the directive to remove the error, but first determine why the directive is there.

(104) preprocessor #assert failure (Preprocessor)
The argument to a preprocessor `#assert` directive has evaluated to zero. This is a programmer induced error.

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm (Preprocessor)
A `#endasm` operator has been encountered, but there was no previous matching `#asm`, for example:

```
void cleardog(void)
{
    clrwdt
    #endasm  /* in-line assembler ends here, only where did it begin? */
}
```

(106) nested #asm directives (Preprocessor)
It is not legal to nest `#asm` directives. Check for a missing or misspelled `#endasm` directive, for example:

```
#asm
    MOVE  r0, #0aah
#asm      ; previous #asm must be closed before opening another
    SLEEP
#endasm
```

(107) illegal # directive “*” (Preprocessor, Parser)
The compiler does not understand the `#` directive. It is probably a misspelling of a directive token, for example:

```
#indef DEBUG  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] without an argument (Preprocessor)
The preprocessor directives `#if`, `#ifdef`, and `#ifndef` must have an argument. The argument to `#if` should be an expression, while the argument to `#ifdef` or `#ifndef` should be a single name, for example:

```
#if            /* oops -- no argument to check */
output = 10;
#else
output = 20;
#endif
```

(109) #include syntax error (Preprocessor)
The syntax of the filename argument to `#include` is invalid. The argument to `#include` must be a valid file name, either enclosed in double quotes `" "` or angle brackets `< >`. Spaces should not be included and the closing quote or bracket must be present. There should be nothing else on the line other than comments, for example:

```
#include stdio.h  /* oops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]] (Preprocessor)
CPP should be invoked with at most two file arguments. Contact Microchip Technical Support if the preprocessor is being executed by a compiler driver.

(111) redefining preprocessor macro “*” (Preprocessor)
The macro specified is being redefined to something different than the original definition. If you want to deliberately redefine a macro, use #undef first to remove the original definition, for example:

```
#define ONE 1
/* elsewhere: */
/* Is this correct? It will overwrite the first definition. */
#define ONE one
```

(112) #define syntax error (Preprocessor)
A macro definition has a syntax error. This could be due to a macro or formal parameter name that does not start with a letter or a missing closing parenthesis,), for example:

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in preprocessor macro body (Preprocessor, Assembler)
A macro definition contains a string that lacks a closing quote.

(114) illegal #undef argument (Preprocessor)
The argument to `#undef` must be a valid name. It must start with a letter, for example:

```
#undef 6YYY  /* this isn’t a valid symbol name */
```

(115) recursive preprocessor macro definition of “*” defined by “*” (Preprocessor)
The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself.

(116) end of file within preprocessor macro argument from line * (Preprocessor)
A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started, for example:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* oops -- where is the closing bracket? */
```

(117) misplaced constant in #if (Preprocessor)
A constant in a `#if` expression should only occur in syntactically correct places. This error is probably caused by omission of an operator, for example:

```
#if FOO BAR  /* oops -- did you mean: #if FOO == BAR ? */
```

(118) stack overflow processing #if expression (Preprocessor)
The preprocessor filled up its expression evaluation stack in a `#if` expression. Simplify the expression – it probably contains too many parenthesized subexpressions.

(119) invalid expression in #if line (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(120) operator “*” in incorrect context (Preprocessor)
An operator has been encountered in a `#if` expression that is incorrectly placed (two binary operators are not separated by a value), for example:

```
#if FOO * % BAR == 4  /* what is "* %" ? */
#define BIG
#endif
```

(121) expression stack overflow at operator “*” (Preprocessor)
Expressions in `#if` lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

(122) unbalanced parenthesis at operator “*” (Preprocessor)
The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesizing, for example:

```
#if ((A) + (B)  /* oops -- a missing ), I think */
#define ADDED
#endif
```

(123) misplaced “?” or “:”; previous operator is “*” (Preprocessor)
A colon operator has been encountered in a `#if` expression that does not match up with a corresponding `?` operator, for example:

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) illegal character “*” in #if (Preprocessor)
There is a character in a `#if` expression that should not be there. Valid characters are the letters, digits and those comprising the acceptable operators, for example:

```
#if YYY  /* what are these characters doing here? */
int m;
#endif
```

(125) illegal character (* decimal) in #if (Preprocessor)
There is a non-printable character in a `#if` expression that should not be there. Valid characters are the letters, digits and those comprising the acceptable operators, for example:

```
#if ^S YYY  /* what is this control characters doing here? */
int m;
#endif
```

(126) strings can’t be used in #if (Preprocessor)
The preprocessor does not allow the use of strings in `#if` expressions, for example:

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if (Preprocessor)
The `defined()` pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter and should be enclosed in parentheses, for example:

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
input = read();
#endif
```

(128) illegal operator in #if (Preprocessor)
A `#if` expression has an illegal operator. Check for correct syntax, for example:

```
#if FOO = 6  /* oops -- should that be: #if FOO == 5 ? */
```

(129) unexpected "\" in #if (Preprocessor)
The backslash is incorrect in the `#if` statement, for example:

```
#if FOO == \34
#define BIG
#endif
```

(130) unknown type "*" in #[el]if sizeof() (Preprocessor)
An unknown type was used in a preprocessor `sizeof()`. The preprocessor can only evaluate `sizeof()` with basic types, or pointers to basic types, for example:

```
#if sizeof(unt) == 2  /* should be: #if sizeof(int) == 2 */
i = 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof() (Preprocessor)
The preprocessor found an illegal type combination in the argument to `sizeof()` in a `#if` expression, for example:

```
/* To sign, or not to sign, that is the error. */
#if sizeof(signed unsigned int) == 2
i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof() (Preprocessor)
`sizeof()` was used in a preprocessor `#if` expression, but no type was specified. The argument to `sizeof()` in a preprocessor expression must be a valid simple type, or pointer to a simple type, for example:

```
#if sizeof()  /* oops -- size of what? */
i = 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof() (Preprocessor)
The preprocessor has made an internal error in evaluating a `sizeof()` expression. Check for a malformed type specifier. This is an internal error. Contact Microchip Technical Support with details.

(134) syntax error in #[el]if sizeof() (Preprocessor)
The preprocessor found a syntax error in the argument to `sizeof()` in a `#if` expression. Probable causes are mismatched parentheses and similar things, for example:

```
#if sizeof(int == 2)  // oops - should be: #if sizeof(int) == 2
i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if (Preprocessor)
The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error. Contact Microchip Technical Support with details.

(137) strange character “*” after ## (Preprocessor)
A character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Because the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, for example:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(138) strange character (*) after ## (Preprocessor)
An unprintable character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Because the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, for example:

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(139) end of file in comment (Preprocessor)
End of file was encountered inside a comment. Check for a missing closing comment flag, for example:

```
/* Here the comment begins. I’m not sure where I end, though
}
```

(140) can’t open * file “*”: * (Driver, Preprocessor, Code Generator, Assembler)
The command file specified could not be opened for reading. Confirm the spelling and path of the file specified on the command line, for example:

```
xc8 @communds
```

should that be:

```
xc8 @commands
```

(141) can’t open * file “*”: * (Any)
An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if blocks (Preprocessor)
`#if`, `#ifdef`, etc., blocks can only be nested to a maximum of 32.

(146) #include filename too long (Preprocessor)
A filename constructed while looking for an include file has exceeded the length of an internal buffer. Because this buffer is 4096 bytes long, this is unlikely to happen.

(147) too many #include directories specified (Preprocessor)
A maximum of 7 directories can be specified for the preprocessor to search for include files. The number of directories specified with the driver is too many.

(148) too many arguments for preprocessor macro (Preprocessor)
A macro can only have up to 31 parameters, per the C Standard.

(149) preprocessor macro work area overflow (Preprocessor)
The total length of a macro expansion has exceeded the size of an internal table. This table is normally 32768 bytes long. Thus any macro expansion must not expand to a total of more than 32K bytes.

(150) illegal "__" preprocessor macro "*" (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(151) too many arguments in preprocessor macro expansion (Preprocessor)
There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

(152) bad dp/nargs in openpar(): c = * (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(153) out of space in preprocessor macro * argument expansion (Preprocessor)
A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

(155) work buffer overflow concatenating "*" (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(156) work buffer "*" overflow (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(157) can't allocate * bytes of memory (Code Generator, Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(158) invalid disable in preprocessor macro "*" (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(159) too many calls to unget() (Preprocessor)
This is an internal compiler error. Contact Microchip Technical Support with details.

(161) control line "*" within preprocessor macro expansion (Preprocessor)
A preprocessor control line (one starting with a `#`) has been encountered while expanding a macro. This should not happen.

(162) #warning: * (Preprocessor, Driver)
This warning is either the result of user-defined `#warning` preprocessor directive, or the driver encountered a problem reading the map file. If the latter, contact Microchip Technical Support with details

(163) unexpected text in control line ignored (Preprocessor)
This warning occurs when extra characters appear on the end of a control line. The extra text will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the text as a comment, for example:

```
#if defined(END)
#define NEXT
#endif END     /* END would be better in a comment here */
```

(164) #include filename "*" was converted to lower case (Preprocessor)
The `#include` file name had to be converted to lowercase before it could be opened, for example:

```
#include <STDIO.H>  /* oops -- should be: #include <stdio.h> */
```

(165) #include filename "*" does not match actual name (check upper/lower case) (Preprocessor)
In Windows versions this means the file to be included actually exists and is spelled the same way as the `#include` filename; however, the case of each does not exactly match. For example, specifying `#include "code.c"` will include `Code.c`, if it is found. In Linux versions this warning could occur if the file wasn't found.

(166) too few values specified with option "*" (Preprocessor)
The list of values to the preprocessor (CPP) `-S` option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver. The values passed to this option represent the sizes of `char`, `short`, `int`, `long`, `float` and `double` types.

(167) too many values specified with -S option; "*" unused Preprocessor)
There were too many values supplied to the `-S` preprocessor option. See message 166.

(168) unknown option "*" (Any)
The option given to the component which caused the error is not recognized.

(169) strange character (*) after ## (Preprocessor)
There is an unexpected character after `#`.

(170) symbol "*" in undef was never defined (Preprocessor)
The symbol supplied as argument to `#undef` was not already defined. This warning can be disabled with some compilers. This warning can be avoided with code like:

```
#ifdef SYM
#undef SYM  /* only undefine if defined */
#endif
```

(171) wrong number of preprocessor macro arguments for "*" (* instead of *) (Preprocessor)
A macro has been invoked with the wrong number of arguments, for example:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)        /* oops -- only two arguments required */
```

(172) formal parameter expected after # (Preprocessor)
The stringization operator `#` (not to be confused with the leading `#` used for preprocessor control lines) must be followed by a formal macro parameter, for example:

```
#define str(x) #y  /* oops -- did you mean x instead of y? */
```

If you need to stringize a token, you will need to define a special macro to do it, for example:

```
#define __mkstr__(x) #x
```

then use `__mkstr__(`*`token`*`)` wherever you need to convert *`token`* into a string.

(173) undefined symbol "*" in #if; 0 used (Preprocessor)
A symbol on a `#if` expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero. This warning can be disabled with some compilers. Example:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
#define GOOD
#endif
```

(174) multi-byte constant “*” isn’t portable (Preprocessor)
Multi-byte constants are not portable; and will be rejected by later passes of the compiler, for example:

```
#if CHAR == 'ab'
#define MULTI
#endif
```

(175) division by zero in #if; zero result assumed (Preprocessor)
Inside a `#if` expression, there is a division by zero which has been treated as yielding zero, for example:

```
#if foo/0  /* divide by 0: was this what you were intending? */
int a;
#endif
```

(176) missing newline (Preprocessor)
A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

(177) symbol “*” in -U option was never defined (Preprocessor)
A macro name specified in a `-U` option to the preprocessor was not initially defined, and thus cannot be undefined.

(179) nested comments (Preprocessor)
This warning is issued when nested comments are found. A nested comment can indicate that a previous closing comment marker is missing or malformed, for example:

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* next comment:
hey, where did this line go? */
```

(180) unterminated comment in included file (Preprocessor)
Comments begun inside an included file must end inside the included file.

(181) non-scalar types can’t be converted to other types (Parser)
You cannot convert a structure, union, or array to another type, for example:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types (Parser)
This expression implies a conversion between incompatible types, i.e., a conversion of a structure type into an integer, for example:

```
struct LAYOUT layout;
int i;
layout = i;          /* int cannot be converted to struct */
```

Note that even if a structure only contains an `int` , for example, it cannot be assigned to an `int` variable and vice versa.

(183) function or function pointer required (Parser)
Only a function or function pointer can be the subject of a function call, for example:

```
int a, b, c, d;
a = b(c+d);   /* b is not a function -- did you mean a = b*(c+d) ? */
```

(184) calling an interrupt function is illegal (Parser)
A function-qualified interrupt cannot be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an interrupt function has special function entry and exit code that is appropriate only for calling from an interrupt. An interrupt function can call other non-interrupt functions.

(185) function does not take arguments (Parser, Code Generator)
This function has no parameters, but it is called here with one or more arguments, for example:

```
int get_value(void);
int main(void)
{
int input;
input = get_value(6);  /* oops --
parameter should not be here */
}
```

(186) too many function arguments (Parser)
This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);      /* call has too many arguments */
```

(187) too few function arguments (Parser)
This function requires more arguments than are provided in this call, for example:

```
void add(int a, int b);
add(5);                 /* this call needs more arguments */
```

(188) constant expression required (Parser)
In this context an expression is required that can be evaluated to a constant at compile time, for example:

```
int a;
switch(input) {
case a:  /* oops!
cannot use variable as part of a case label */
input++;
}
```

(189) illegal type for array dimension (Parser)
An array dimension must be either an integral type or an enumerated value.

```
int array[12.5];  /* oops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression (Parser)
An index expression must be either integral or an enumerated value, for example:

```
int i, array[10];
i = array[3.5];   /* oops --
exactly which element do you mean? */
```

(191) cast type must be scalar or void (Parser)
A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e., not an array or a structure) or the type `void`, for example:

```
lip = (long [])input;  /* oops -- possibly: lip = (long *)input */
```

(192) undefined identifier “*” (Parser)
This symbol has been used in the program, but has not been defined or declared. Check for spelling errors if you think it has been defined.

(193) not a variable identifier “*” (Parser)
This identifier is not a variable; it can be some other kind of object, i.e., a label.

(194) “)” expected (Parser)
A closing parenthesis,), was expected here. This can indicate you have left out this character in an expression, or you have some other syntax error. The error is flagged on the line at which the code first starts to make no sense. This can be a statement following the incomplete expression, for example:

```
if(a == b  /* the closing parenthesis is missing here */
b = 0;   /* the error is flagged here */
```

(195) expression syntax (Parser)
This expression is badly formed and cannot be parsed by the compiler, for example:

```
a /=% b;  /* oops -- possibly that should be: a /= b; */
```

(196) struct/union required (Parser)
A structure or union identifier is required before a dot “.” , for example:

```
int a;
a.b = 9;  /* oops -- a is not a structure */
```

(197) struct/union member expected (Parser)
A structure or union member name must follow a dot “.” or an arrow (“->”).

(198) undefined struct/union “*” (Parser)
The specified structure or union tag is undefined, for example:

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) logical type required (Parser)
The expression used as an operand to `if`, `while` statements or to boolean operators like `!` and `&&` must be a scalar integral type, for example:

```
struct FORMAT format;
if(format)            /* this operand must be a scaler type */
    format.a = 0;
```

(200) taking the address of a register variable is illegal (Parser)
A variable declared register cannot have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the `&` operator, for example:

```
int * proc(register int in)
{
    int * ip = &in;
    /* oops -- in cannot have an address to take */
    return ip;
}
```

(201) taking the address of this object is illegal (Parser)
The expression which was the operand of the `&` operator is not one that denotes memory storage (“an lvalue”) and therefore its address cannot be defined, for example:

```
ip = &8;  /* oops -- you cannot take the address of a literal */
```

(202) only lvalues can be assigned to or modified (Parser)
Only an lvalue (i.e., an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified, for example:

```
int array[10];
int * ip;
char c;
array = ip;   /* array is not a variable, it cannot be written to */
```

A typecast does not yield an lvalue, for example:

```
/* the contents of c cast to int is only a intermediate value */
(int)c = 1;
```

However, you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on bit variable (Parser)
Not all operations on bit variables are supported. This operation is one of those, for example:

```
bit   b;
int * ip;
ip = &b;  /* oops -- cannot take the address of a bit object */
```

(204) void function can’t return a value (Parser)
A void function cannot return a value. Any return statement should not be followed by an expression, for example:

```
void run(void)
{
    step();
    return 1;  /* either run should not be void, or remove the 1 */
}
```

(205) integral type required (Parser)
This operator requires operands that are of integral type only.

(206) illegal use of void expression (Parser)
A `void` expression has no value and therefore you cannot use it anywhere an expression with a value is required, i.e., as an operand to an arithmetic operator.

(207) simple type required for “*” (Parser)
A simple type (i.e., not an array or structure) is required as an operand to this operator.

(208) operands of “*” not same type (Parser)
The operands of this operator are of different pointers, for example:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;  /* result of ? : will be int * or char * */
```

Possibly, you meant something like:

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict (Parser)
The operands of this operator are of incompatible types.

(210) bad size list (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(211) taking sizeof bit is illegal (Parser)
It is illegal to use the `sizeof()` operator with the C `__bit` type. When used against a type, the `sizeof()` operator gives the number of bytes required to store an object that type. Therefore its usage with the `__bit` type make no sense and it is an illegal operation.

(212) missing number after pragma “pack” (Parser)
The `pragma pack` requires a decimal number as argument. This specifies the alignment of each member within the structure. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries, for example:

```
#pragma pack  /* what is the alignment value */
```

Possibly, you meant something like:

```
#pragma pack 2
```

(214) missing number after pragma “interrupt_level” (Parser)
The `pragma interrupt_level` requires an argument to indicate the interrupt level. It will be the value 1 for mid-range devices, or 1 or 2 or PIC18 devices.

(215) missing argument to pragma “switch” (Parser)
The `pragma switch` requires an argument of auto, direct or simple, for example:

```
#pragma switch  /* oops -- this requires a switch mode */
```

Possibly, you meant something like:

```
#pragma switch simple
```

(216) missing argument to pragma “psect” (Parser)
The `pragma psect` requires an argument of the form *`oldname = newname`* where *`oldname`* is an existing psect name known to the compiler and *`newname`* is the desired new name, for example:

```
#pragma psect  /* oops -- this requires an psect to redirect */
```

Possibly, you meant something like:

```
#pragma psect text=specialtext
```

(218) missing name after pragma “inline” (Parser)
The `inline pragma` expects the name of a function to follow. The function name must be recognized by the code generator for it to be expanded; other functions are not altered, for example:

```
#pragma inline  /* what is the function name? */
```

Possibly, you meant something like:

```
#pragma inline memcpy
```

(219) missing name after pragma “printf_check” (Parser)
The `printf_check pragma` expects the name of a function to follow. This specifies printf-style format string checking for the function, for example:

```
#pragma printf_check  /* what function is to be checked? */
```

Possibly, you meant something like:

```
#pragma printf_check sprintf
```

Pragmas for all the standard printf-like function are already contained in `<stdio.h>`.

(220) exponent expected (Parser)
A floating-point constant must have at least one digit after the `e` or `E`, for example:

```
float f;
f = 1.234e;  /* oops -- what is the exponent? */
```

(221) hexadecimal digit expected (Parser)
After `0x` should follow at least one of the HEX digits 0-9 and A-F or a-f, for example:

```
a = 0xg6;  /* oops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected (Parser)
A binary digit was expected following the `0b` format specifier, for example:

```
i = 0bf000;  /* oops -- f000 is not a base two value */
```

(223) digit out of range (Parser, Assembler)
A digit in this number is out of range of the radix for the number, i.e., using the digit 8 in an octal number, or HEX digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a HEX number starts with `0X` or `0x`. For example:

```
int a = 058; /* leading 0 implies octal which has digits 0 - 7 */
```

(224) illegal “#” directive (Parser)
An illegal `#` preprocessor has been detected. Likely, a directive has been misspelled in your code somewhere.

(225) missing character in character constant (Parser)
The character inside the single quotes is missing, for example:

```
char c = ';  /* the character value of what? */
```

(226) char const too long (Parser)
A character constant enclosed in single quotes cannot contain more than one character, for example:

```
c = ’12’;  /* oops -- only one character can be specified */
```

(227) “.” expected after “..” (Parser)
The only context in which two successive dots can appear is as part of the ellipsis symbol, which must have 3 dots (an ellipsis is used in function prototypes to indicate a variable number of parameters).

Either .. was meant to be an ellipsis symbol which would require you to add an extra dot, or it was meant to be a structure member operator which would require you to remove one dot.

(228) illegal character (*) (Parser)
This character is illegal in the C code. Valid characters are the letters, digits and those comprising the acceptable operators, for example:

```
c = a;  /* oops -- did you mean c = 'a'; ? */
```

(229) unknown qualifier "*" given to -A (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(230) missing argument to -A (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(231) unknown qualifier "*" given to -I (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(232) missing argument to -I (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(233) bad -Q option "*" (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(234) close error (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(236) simple integer expression required (Parser)
A simple integral expression is required after the `__at()` operator, used to associate an absolute address with a variable, for example:

```
int address;
char LOCK __at(address);
```

(237) function "*" redefined (Parser)
More than one definition for a function has been encountered in this module. Function overloading is illegal, for example:

```
int twice(int a)
{
    return a*2;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
    return a*2;
}
```

(238) illegal initialization (Parser)
You cannot initialize a `typedef` declaration, because it does not reserve any storage that can be initialized, for example:

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(239) identifier “*” redefined (from line *) (Parser)
This identifier has already been defined in the same scope. It cannot be defined again, for example:

```
int a;  /* a filescope variable called "a" */
int a;  /* attempting to define another of the same name */
```

Note that variables with the same name, but defined with different scopes, are legal; but; not recommended.

(240) too many initializers (Parser)
There are too many initializers for this object. Check the number of initializers against the object definition (array or structure), for example:

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(241) initialization syntax (Parser)
The initialization of this object is syntactically incorrect. Check for the correct placement and number of braces and commas, for example:

```
int iarray[10] = {{’a’, ’b’, ’c’};
/* oops -- one two many {s */
```

(242) illegal type for switch expression (Parser)
A `switch()` operator must have an expression that is either an integral type or an enumerated value, e.g:

```
double d;
switch(d) {  /* oops -- this must be integral */
case ’1.0’:
    d = 0;
}
```

(243) inappropriate break/continue (Parser)
A `break` or `continue` statement has been found that is not enclosed in an appropriate control structure. A continue can only be used inside a `while`, `for`, or `do while` loop, while `break` can only be used inside those loops or a `switch` statement, for example:

```
switch(input) {
case 0:
    if(output == 0)
        input = 0xff;
    } /* oops! this should not be here; it closed the switch */
    break;     /* this should be inside the switch */
```

(244) “default” case redefined (Parser)
Only one `default` label is allowed to be in a `switch()` statement. You have more than one, for example:

```
switch(a) {
default:     /* if this is the default case... */
    b = 9;
    break;
default:     /* then what is this? */
    b = 10;
    break;
```

(245) “default” case not in switch (Parser)
A label has been encountered called `default`, but it is not enclosed by a `switch` statement. A default label is only legal inside the body of a `switch` statement.

If there is a `switch` statement before this default label, there could be one too many closing braces in the `switch` code. That would prematurely terminate the `switch` statement. See message 246.

(246) case label not in switch (Parser)
A `case` label has been encountered, but there is no enclosing `switch` statement. A `case` label can only appear inside the body of a `switch` statement.

If there is a `switch` statement before this `case` label, there might be one too many closing braces in the `switch` code. That would prematurely terminate the switch statement, for example:

```
switch(input) {
case ’0’:
    count++;
    break;
case ’1’:
    if(count>MAX)
        count= 0;
    }          /* oops -- this shouldn’t be here */
    break;
case ’2’:    /* error flagged here */
```

(247) duplicate label “*” (Parser)
The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label, for example:

```
start:
if(a > 256)
    goto end;
start:           /* error flagged here */
if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) inappropriate “else” (Parser)
An `else` keyword has been encountered that cannot be associated with an `if` statement. This can mean there is a missing brace or other syntactic error, for example:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
    c = 0;
/* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
    c = 0xff;
```

(249) probable missing “}” in previous block (Parser)
The compiler has encountered what looks like a function or other declaration, but the preceding function was ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it might not be the last one, for example:

```
void set(char a)
{
    PORTA = a;
    /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
    PORTA = 0;
}
```

(251) array dimension redeclared (Parser)
An array dimension has been declared as a different non-zero value from its previous declaration. It is acceptable to redeclare the size of an array that was previously declared with a zero dimension; but, not otherwise, for example:

```
extern int array[5];
int array[10];        /* oops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype (Parser)
The argument specified (argument 0 is the left most argument) of this function definition does not agree with a previous prototype for this function, for example:

```
/* this is supposedly calc's prototype */
extern int calc(int, int);
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
    return sin(b/a);
}
```

(253) argument list conflicts with prototype (Parser)
The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

```
extern int calc(int);   /* this is supposedly calc's prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
    return a + b;
}
```

(254) undefined *: "*" (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(255) not a member of the struct/union "*" (Parser)
This identifier is not a member of the structure or union type with which it used here, for example:

```
struct {
    int a, b, c;
} data;
if(data.d)    /* oops -- there is no member d in this structure */
    return;
```

(256) too much indirection (Parser)
A pointer declaration can only have 16 levels of indirection.

(257) only "register" storage class allowed (Parser)
The only storage class allowed for a function parameter is `register`, for example:

```
void process(static int input)
```

(258) duplicate qualifier (Parser)
There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a `typedef`. Remove the redundant qualifier. For example:

```
typedef volatile int vint;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) object can't be qualified both far and near (Parser)
It is illegal to qualify a type as both `far` and `near`, for example:

```
far near int spooky;  /* oops -- choose far or near, not both */
```

(260) undefined enum tag "*" (Parser)
This `enum` tag has not been defined, for example:

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) struct/union member "*" redefined (Parser)
This name of this member of the `struct` or `union` has already been used in this `struct` or `union`, for example:

```
struct {
    int a;
    int b;
    int a;  /* oops -- a different name is required here */
} input;
```

(262) struct/union "*" redefined (Parser)
A structure or union has been defined more than once, for example:

```
struct {
    int a;
} ms;
struct {
    int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members can't be functions (Parser)
A member of a structure or a union cannot be a function. It could be a pointer to a function, for example:

```
struct {
    int a;
    int get(int);  /* should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type (Parser)
A bit-field can only have a type of int (or unsigned), for example:

```
struct FREG {
    char b0:1;   /* these must be part of an int, not char */
    char   :6;
    char b7:1;
} freg;
```

(265) integer constant expected (Parser)
A colon appearing after a member name in a structure declaration indicates that the member is a bit-field. An integral constant must appear after the colon to define the number of bits in the bit-field, for example:

```
struct {
    unsigned first:  /* oops -- should be: unsigned first; */
    unsigned second;
} my_struct;
```

If this was meant to be a structure with bit-fields, then the following illustrates an example:

```
struct {
    unsigned first : 4;  /* 4 bits wide */
```

```
    unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) storage class illegal (Parser)
A structure or union member cannot be given a storage class. Its storage class is determined by the storage class of the structure, for example:

```
struct {
    /* no additional qualifiers can be present with members */
    static int first;
} ;
```

(267) bad storage class (Code Generator)
The code generator has encountered a variable definition whose storage class is invalid, for example:

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters cannot be static */
{
    return foo * a;
}
```

(268) inconsistent storage class (Parser)
A declaration has conflicting storage classes. Only one storage class should appear in a declaration, for example:

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type (Parser)
Only one basic type can appear in a declaration, for example:

```
int float input;  /* is it int or float? */
```

(270) variable can’t have storage class “register” (Parser)
Only function parameters or auto variables can be declared using the `register` qualifier, for example:

```
register int gi;      /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
    return input + gi;
}
```

(271) type can’t be long (Parser)
Only `int` and `double` can be qualified with long.

```
long char lc;  /* what? */
```

(272) type can’t be short (Parser)
Only `int` can be modified with `short`, for example:

```
short float sf;  /* what? */
```

(273) type can’t be both signed and unsigned (Parser)
The type modifiers `signed` and `unsigned` cannot be used together in the same declaration, as they have opposite meaning, for example:

```
signed unsigned int confused;  /* which is it? */
```

(274) type can't be unsigned (Parser)
A floating-point type cannot be made `unsigned`, for example:

```
unsigned float uf;  /* what? */
```

(275) "..." illegal in non-prototype argument list (Parser)
The ellipsis symbol can only appear as the last item in a prototyped argument list. It cannot appear on its own, nor can it appear after argument names that do not have types; i.e., K&R-style non-prototype function definitions. For example:

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
    int a, b;
{
```

(276) type specifier required for prototyped argument (Parser)
A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) can't mix prototyped and non-prototyped arguments (Parser)
A function declaration can only have all prototyped arguments (i.e., with types inside the parentheses) or all K&R style arguments (i.e., only names inside the parentheses and the argument types in a declaration list before the start of the function body), for example:

```
int plus(int a, b)  /* oops -- a is prototyped, b is not */
int b;
{
    return a + b;
}
```

(278) argument "*" redeclared (Parser)
The specified argument is declared more than once in the same argument list, for example:

```
/* cannot have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal (Parser)
A function argument cannot have an initializer in a declaration. The initialization of the argument happens when the function is called and a value is provided for the argument by the calling function, for example:

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal (Parser)
You cannot define an array of functions. You can, however, define an array of pointers to functions, for example:

```
int * farray[]();  /* oops -- should be: int (* farray[])(); */
```

(281) functions can't return functions (Parser)
A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: `int (* (`*name*`()))()`. Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

(282) functions can't return arrays (Parser)
A function can return only a scalar (simple) type or a structure. It cannot return an array.

(283) dimension required (Parser)
Only the most significant (i.e., the first) dimension in a multi-dimension array cannot be assigned a value. All succeeding dimensions must be present as a constant expression, for example:

```
/* This should be, for example: int arr[][7] */
int get_element(int arr[2][])
{
    return array[1][6];
}
```

(284) invalid dimension (Parser)
The array dimension specified is not valid. It must be larger than 0.

```
int array[0];  // oops -- you cannot have an array of size 0
```

(285) no identifier in declaration (Parser)
The identifier is missing in this declaration. This error can also occur when the compiler has been confused by such things as missing closing braces, for example:

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) declarator too complex (Parser)
This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

(287) arrays of bits or pointers to bit are illegal (Parser)
It is not legal to have an array of `__bit` objects, or a pointer to `bit` variable, for example:

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) the type 'void' is applicable only to functions (Parser)
A variable cannot be `void`. Only a function can be void, for example:

```
int a;
void b;  /* this makes no sense */
```

(289) the specifier 'interrupt' is applicable only to functions (Parser)
The qualifier interrupt cannot be applied to anything except a function, for example:

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(290) illegal function qualifier(s) (Parser)
A qualifier has been applied to a function which makes no sense in this context. Some qualifier only make sense when used with an lvalue, i.e., `const` or `volatile`. This can indicate that you have forgotten a star, `*`, that is indicating that the function should return a pointer to a qualified object, for example:

```
const char ccrv(void) /* const * char ccrv(void) perhaps? */
{                     /* error flagged here */
    return ccip;
}
```

(291) K&R identifier “*” not an argument (Parser)
This identifier, that has appeared in a K&R style argument declarator, is not listed inside the parentheses after the function name, for example:

```
int process(input)
int unput;        /* oops -- that should be int input; */
{
}
```

(292) a function is not a valid parameter type (Parser)
A function parameter cannot be a function. It can be a pointer to a function, so perhaps a `*` has been omitted from the declaration.

(293) bad size in index_type() (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(294) can’t allocate * bytes of memory (Code Generator, Hexmate)
This is an internal compiler error. Contact Microchip Technical Support with details.

(295) expression too complex (Parser)
This expression has caused overflow of the compiler’s internal stack and should be rearranged or split into two expressions.

(296) out of memory (Objtohex)
This could be an internal compiler error. Contact Microchip Technical Support with details.

(297) bad argument (*) to tysize() (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(298) end of file in #asm (Preprocessor)
An end of file has been encountered inside a `#asm` block. This probably means the `#endasm` is missing or misspelled, for example:

```
#asm
MOV  r0, #55
MOV  [r1], r0
}                /* oops -- where is the #endasm */
```

(300) unexpected end of file (Parser)
An end-of-file in a C module was encountered unexpectedly, for example:

```
int main(void)
{
    init();
    run();    /* is that it? What about the close brace */
```

(301) end of file on string file (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(302) can’t reopen “*”: * (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(303) can’t allocate * bytes of memory (line *) (Parser)
The parser was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

(306) can't allocate * bytes of memory for * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(307) too many qualifier names (Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(308) too many case labels in switch (Code Generator)
There are too many case labels in this `switch` statement. The maximum allowable number of case labels in any one `switch` statement is 511.

(309) too many symbols (Assembler, Parser)
There are too many symbols for the assembler's symbol table. Reduce the number of symbols in your program.

(310) "]" expected (Parser)
A closing square bracket was expected in an array declaration or an expression using an array index, for example:

```
process(carray[idx);  /* oops --
should be: process(carray[idx]); */
```

(311) closing quote expected (Parser)
A closing quote was expected for the indicated string.

(312) "*" expected (Parser)
The indicated token was expected by the parser.

(313) function body expected (Parser)
Where a function declaration is encountered with K&R style arguments (i.e., argument names; but, no types inside the parentheses) a function body is expected to follow, for example:

```
/* the function block must follow, not a semicolon */
int get_value(a, b);
```

(314) ";" expected (Parser)
A semicolon is missing from a statement. A close brace or keyword was found following a statement with no terminating semicolon , for example:

```
while(a) {
    b = a--  /* oops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceding a close brace or keyword typically results in some other error being issued for the following code which the parser assumes to be part of the original statement.

(315) "{" expected (Parser)
An opening brace was expected here. This error can be the result of a function definition missing the opening brace, for example:

```
/* oops! no opening brace after the prototype */
void process(char c)
    return max(c, 10) * 2; /* error flagged here */
}
```

(316) "}" expected (Parser)
A closing brace was expected here. This error can be the result of a initialized array missing the closing brace, for example:

```
char carray[4] = { 1, 2, 3, 4;  /* oops -- no closing brace */
```

(317) "(" expected (Parser)
An opening parenthesis , `(`, was expected here. This must be the first token after a `while` , `for` , `if` , `do` or `asm` keyword, for example:

```
if a == b  /* should be: if(a == b) */
b = 0;
```

(318) string expected (Parser)
The operand to an `asm` statement must be a string enclosed in parentheses, for example:

```
asm(nop);  /* that should be asm("nop");
```

(319) while expected (Parser)
The keyword `while` is expected at the end of a `do` statement, for example:

```
do {
    func(i++);
}             /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
    end();
```

(320) ":" expected (Parser)
A colon is missing after a `case` label, or after the keyword `default`. This often occurs when a semicolon is accidentally typed instead of a colon, for example:

```
switch(input) {
case 0;          /* oops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected (Parser)
An identifier denoting a label must appear after `goto`, for example:

```
if(a)
    goto 20;
    /* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or "{" expected (Parser)
After the keyword `enum`, must come either an identifier that is, or will be, defined as an `enum` tag, or an opening brace, for example:

```
enum 1, 2;  /* should be, for example: enum {one=1, two }; */
```

(323) struct/union tag or "{" expected (Parser)
An identifier denoting a structure or union or an opening brace must follow a `struct` or `union` keyword, for example:

```
struct int a;  /* this is not how you define a structure */
```

You might mean something like:

```
struct {
    int a;
} my_struct;
```

(324) too many arguments for printf-style format string (Parser)
There are too many arguments for this format string. This is harmless, but can represent an incorrect format string, for example:

```
/* oops -- missed a placeholder? */
printf("%d - %d", low, high, median);
```

(325) error in printf-style format string (Parser)
There is an error in the format string here. The string has been interpreted as a `printf()` style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behavior at runtime, for example:

```
printf("%l", lll);  /* oops -- possibly: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string (Parser)
A `long` argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, for example:

```
printf("%lx", 2);  // possibly you meant: printf("%lx", 2L);
```

(327) long long int argument required in printf-style format string (Parser)
A `long long` argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, for example:

```
printf("%llx", 2);  // possibly you meant: printf("%llx", 2LL);
```

(328) int argument required in printf-style format string (Parser)
An integral argument is required for this printf-style format specifier. Check the number and order of format specifiers and corresponding arguments, for example:

```
printf("%d", 1.23); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string (Parser)
The printf format specifier corresponding to this argument is `%f` or similar, and requires a floating-point expression. Check for missing or extra format specifiers or arguments to printf.

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required in printf-style format string (Parser)
A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

(331) too few arguments for printf-style format string (Parser)
There are too few arguments for this format string. This would result in a garbage value being printed or converted at runtime, for example:

```
printf("%d - %d", low); /* oops! where is the other value to print? */
```

(332) “interrupt_level” should be 0 to 7 (Parser)
The `pragma interrupt_level` must have an argument from 0 to 7; however, mid-range devices only use level 1. PIC18 devices can use levels 1 or 2. For example:

```
#pragma interrupt_level  9 /* oops -- the level is too high */
void interrupt isr(void)
{
    /* isr code goes here */
}
```

(333) unrecognized qualifier name after “strings” (Parser)
The `pragma strings` was passed a qualifier that was not identified, for example:

```
/* oops -- should that be #pragma strings const ? */
#pragma strings cinst
```

(334) unrecognized qualifier name after “printf_check” (Parser)
The `#pragma printf_check` was passed a qualifier that could not be identified, for example:

```
/* oops -- should that be const not cinst? */
#pragma printf_check(printf) cinst
```

(335) unknown pragma “*” (Parser)
An unknown `pragma` directive was encountered, for example:

```
#pragma rugsused myFunc w  /* I think you meant regsused */
```

(336) string concatenation across lines (Parser)
Strings on two lines will be concatenated. Check that this is the desired result, for example:

```
char * cp = "hi"
"there";    /* this is okay, but is it what you had intended? */
```

(337) line does not have a newline on the end (Parser)
The last line in the file is missing the newline (operating system dependent character) from the end. Some editors will create such files, which can cause problems for include files. The C standard requires all source files to consist of complete lines only.

(338) can’t create * file “*” (Any)
The application tried to create or open the named file, but it could not be created. Check that all file path names are correct.

(339) initializer in extern declaration (Parser)
A declaration containing the keyword `extern` has an initializer. This overrides the `extern` storage class, because to initialize an object it is necessary to define (i.e., allocate storage for) it, for example:

```
extern int other = 99;  /* if it’s extern and not allocated
storage, how can it be initialized? */
```

(340) string not terminated by null character (Parser)
A `char` array is being initialized with a string literal larger than the array. Hence there is insufficient space in the array to safely append a null terminating character, for example:

```
char foo[5] = "12345"; /* the string stored in foo won’t have
a null terminating, i.e.
foo = [’1’, ’2’, ’3’, ’4’, ’5’] */
```

(343) implicit return at end of non-void function (Parser)
A function that has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a `return` statement with a value, or if the function is not to return a value, declare it `void`, for example:

```
int mydiv(double a, int b)
{
    if(b != 0)
        return a/b;   /* what about when b is 0? */
}                 /* warning flagged here */
```

(344) non-void function returns no value (Parser)
A function that is declared as returning a value has a `return` statement that does not specify a return value, for example:

```
int get_value(void)
{
    if(flag)
        return val++;
    return;
    /* what is the return value in this instance? */
}
```

(345) unreachable code (Parser)
This section of code will never be executed, because there is no execution path by which it could be reached, for example:

```
while(1)          /* how does this loop finish? */
    process();
flag = FINISHED;  /* how do we get here? */
```

(346) declaration of “*” hides outer declaration (Parser)
An object has been declared that has the same name as an outer declaration (i.e., one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended, for example:

```
int input;         /* input has filescope */
void process(int a)
{
    int input;       /* local blockscope input */
    a = input;       /* this will use the local variable. Is this right? */
```

(347) external declaration inside function (Parser)
A function contains an `extern` declaration. This is legal but is invariably not desirable as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use, or definition of the `extern` object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behavior of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler’s type checking. As a general rule, always declare `extern` variables and functions outside any other functions. For example:

```
int process(int a)
{
    /* this would be better outside the function */
    extern int away;
    return away + a;
}
```

(348) auto variable “*” should not be qualified (Parser)
An auto variable should not have qualifiers such as `near` or `far` associated with it. Its storage class is implicitly defined by the stack organization. An auto variable can be qualified with `static`, but it is then no longer auto.

(349) non-prototyped function declaration for “*” (Parser)
A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions, for example:

```
int process(input)
int input;          /* warning flagged here */
{
}
```

This would be better written:

```
int process(int input)
{
}
```

(350) unused * “*” (from line *) (Parser)
The indicated object was never used in the function or module being compiled. Either this object is redundant, or the code that was meant to use it was excluded from compilation or misspelled the name of the object. Note that the symbols `rcsid` and `sccsid` are never reported as being unused.

(352) float parameter coerced to double (Parser)
Where a non-prototyped function has a parameter declared as `float`, the compiler converts this to a `double`. This is because the default C type conversion conventions provide that when a floating-point number is passed to a non-prototyped function, it is converted to `double`. It is important that the function declaration be consistent with this convention, for example:

```
double inc_flt(f)  /* f will be converted to double */
float f;           /* warning flagged here */
{
    return f * 2;
}
```

(353) sizeof external array “*” is zero (Parser)
The size of an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

(354) possible pointer truncation (Parser)
A pointer qualified far has been assigned to a default pointer, or a pointer qualified near, or a default pointer has been assigned to a pointer qualified near. This can result in truncation of the pointer and loss of information, depending on the memory model in use.

(355) implicit signed to unsigned conversion (Parser)
A signed number is being assigned or otherwise converted to a larger unsigned type. Under the ANSI C “value preserving” rules, this will result in the signed value being first sign-extended to a signed number the size of the target type, then converted to unsigned (which involves no change in bit pattern). An unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, for example:

```
signed char sc;
unsigned int ui;
ui = sc;    /* if sc contains 0xff, ui will contain 0xffff for example */
```

will perform a sign extension of the char variable to the longer type. If you do not want this to take place, use a cast, for example:

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer (Parser)
A floating-point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating-point value. A typecast will make this warning go away.

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)dd;
```

(357) illegal conversion of integer to pointer (Parser)
An integer has been assigned to, or otherwise converted to, a pointer type. This will usually mean that you have used the wrong variable. But if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This can also mean that you have forgotten the `&` address operator, for example:

```
int * ip;
int i;
ip = i;    /* oops -- did you mean ip = &i ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer (Parser)
A pointer has been assigned to, or otherwise converted to, a integral type. This will usually mean that you have used the wrong variable. But if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This can also mean that you have forgotten the `*` dereference operator, for example:

```
int * ip;
int i;
i = ip;    /* oops -- did you mean i = *ip ? */
```

If you do intend to use an expression like this, indicate your intention by a cast:

```
i = (int)ip;
```

(359) illegal conversion between pointer types (Parser)
A pointer of one type (i.e., pointing to a particular kind of object) has been converted into a pointer of a different type. This usually means that you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed, for example:

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

This is a common way of accessing bytes within a multi-byte variable. To indicate that this is the intended operation of the program, use a cast:

```
cp = (char *)&input;  /* that's better */
```

This warning can also occur when converting between pointers to objects that have the same type, but which have different qualifiers, for example:

```
char * cp;
/* yes, but what sort of characters? */
cp = "I am a string of characters";
```

If the default type for string literals is `const char *`, then this warning is quite valid. This should be written:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

Omitting a qualifier from a pointer type is often disastrous and almost certainly not what you intend.

(360) array index out of bounds (Parser)
An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array. This warning will not be issued when accessing an array element via a pointer variable, for example:

```
int i, * ip, input[10];
i = input[-2];           /* oops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];              /* this is okay */
```

(361) function declared implicit int (Parser)
When the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type int, with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined (or at least declared) before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords `extern` or `static`, as appropriate. For example:

```
/* I can prevent an error arising from calls below */
extern void set(long a, int b);
int main(void)
{
    /* at this point, a prototype for set() has already been seen */
    set(10L, 6);
}
```

(362) redundant "&" applied to array (Parser)
The address operator `&` has been applied to an array. Because using the name of an array gives its address anyway, this is unnecessary and has been ignored, for example:

```
int array[5];
int * ip;
/* array is a constant, not a variable; the & is redundant. */
ip = &array;
```

(363) redundant "&" or "*" applied to function address (Parser)
The address operator `&` has been applied to a function. Because using the name of a function gives its address anyway, this is unnecessary and has been ignored, for example:

```
extern void foo(void);
int main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
```

```
    bar = foo;  /* the & is redundant */
}
```

(364) attempt to modify object qualified * (Parser)
Objects declared `const` or `code` cannot be assigned to or modified in any other way by your program. The effect of attempting to modify such an object is compiler specific.

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* oops -- writing to a read-only object */
```

(365) pointer to non-static object returned (Parser)
This function returns a pointer to a non-static (e.g., auto) variable. This is likely to be an error, because the storage associated with automatic variables becomes invalid when the function returns, for example:

```
char * get_addr(void)
{
    char c;
    /* returning this is dangerous; the pointer could be dereferenced */
    return &c;
}
```

(366) operands of “*” not same pointer type (Parser)
The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

(367) identifier is already extern; can’t be static (Parser)
This function was already declared `extern`, possibly through an implicit declaration. It has now been redeclared `static`, but this redeclaration is invalid.

```
int main(void)
{
    /* at this point the compiler assumes set is extern... */
    set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
    PORTA = a + b;
}
```

(368) array dimension on “*[]” ignored (Preprocessor)
An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size can be passed. Either remove the dimension from the parameter, or define the parameter using pointer syntax, for example:

```
/* param should be: "int array[]" or "int *" */
int get_first(int array[10])
{                           /* warning flagged here */
    return array[0];
}
```

(369) signed bitfields not supported (Parser)
Only unsigned bit-fields are supported. If a bit-field is declared to be type `int`, the compiler still treats it as unsigned, for example:

```
struct {
    signed int sign: 1;    /* oops -- this must be unsigned */
    signed int value: 7;
} ;
```

(370) illegal basic type; int assumed (Parser)
The basic type of a cast to a qualified basic type could not be recognized and the basic type was assumed to be int, for example:

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) 'a';
```

(371) missing basic type; int assumed (Parser)
This declaration does not include a basic type, so `int` has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended, for example:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) "," expected (Parser)
A comma was expected here. This could mean you have left out the comma between two identifiers in a declaration list. It can also mean that the immediately preceding type name is misspelled and has been interpreted as an identifier, for example:

```
unsigned char a;
/* thinks: chat & b are unsigned, but where is the comma? */
unsigned chat b;
```

(373) implicit signed to unsigned conversion (Parser)
An `unsigned` type was expected where a `signed` type was given and was implicitly converted to `unsigned`, for example:

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed (Parser)
The basic type of a cast to a qualified basic type was missing and assumed to be `int`., for example:

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(375) unknown FNREC type "*" (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(376) bad non-zero node in call graph (Linker)
The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

(378) can't create * file "*" (Hexmate)
This type of file could not be created. Is the file, or a file by this name, already in use?

(379) bad record type "*" (Linker)
This is an internal compiler error. Ensure that the object file is a valid object file. Contact Microchip Technical Support with details.

(380) unknown record type (*) (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(381) record "*" too long (*) (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(382) incomplete record: type = *, length = * (Dump, Xstrip)
This message is produced by the `dump` or `xstrip` utilities and indicates that the object file is not a valid object file, or that it has been truncated. Contact Microchip Technical Support with details.

(383) text record has length (*) too small (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(384) assertion failed: file *, line *, expression * (Linker, Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(387) illegal or too many -G options (Linker)
There has been more than one linker `-g` option, or the `-g` option did not have any arguments following. The arguments specify how the segment addresses are calculated.

(388) duplicate -M option (Linker)
The map file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of this option is present on the command line.

(389) illegal or too many -O options (Linker)
This linker `-o` flag is illegal, or another `-o` option has been encountered. A `-o` option to the linker must be immediately followed by a filename with no intervening space.

(390) missing argument to -P (Linker)
There have been too many `-p` options passed to the linker, or a `-p` option was not followed by any arguments. The arguments of separate `-p` options can be combined and separated by commas.

(391) missing argument to -Q (Linker)
The `-Q` linker option requires the machine type for an argument.

(392) missing argument to -U (Linker)
The `-U` (undefine) option needs an argument.

(393) missing argument to -W (Linker)
The `-W` option (listing width) needs a numeric argument.

(394) duplicate -D or -H option (Linker)
The symbol file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of either of these options is present on the command line.

(395) missing argument to -J (Linker)
The maximum number of errors before aborting must be specified following the `-j` linker option.

(397) usage: hlink [-options] files.obj files.lib (Linker)
Improper usage of the command-line linker. If you are not invoking the linker directly, this could be an internal compiler error, and you should contact Microchip Technical Support with details.

(398) output file can’t be also an input file (Linker)
The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

(400) bad object code format (Linker)
This is an internal compiler error. The object code format of an object file is invalid. Ensure it is a valid object file. Contact Microchip Technical Support with details.

(402) bad argument to -F (Objtohex)
The `-F` option for objtohex has been supplied an invalid argument. If you are not invoking this tool directly, this is an internal compiler error, and you should contact Microchip Technical Support with details.

(403) bad -E option: “*” (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(404) bad maximum length value to -<digits> (Objtohex)
The first value to the OBJTOHEX *-n,m* HEX length/rounding option is invalid.

(405) bad record size rounding value to -<digits> (Objtohex)
The second value to the OBJTOHEX *-n,m* HEX length/rounding option is invalid.

(406) bad argument to -A (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(407) bad argument to -U (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(408) bad argument to -B (Objtohex)
This option requires an integer argument in either base 8, 10, or 16. If you are not invoking this tool directly, this is an internal compiler error and you should contact Microchip Technical Support with details.

(409) bad argument to -P (Objtohex)
This option requires an integer argument in either base 8, 10, or 16. If you are not invoking this tool directly, this is an internal compiler error and you should contact Microchip Technical Support with details.

(410) bad combination of options (Objtohex)
The combination of options supplied to OBJTOHEX is invalid.

(412) text does not start at 0 (Objtohex)
Code in some things must start at zero. Here it doesn’t.

(413) write error on “*” (Assembler, Linker, Cromwell)
A write error occurred on the named file. This probably means you have run out of disk space.

(414) read error on “*” (Linker)
The linker encountered an error trying to read this file.

(415) text offset too low in COFF file (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(416) bad character (*) in extended TEKHEX line (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(417) seek error in “*” (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(418) image too big (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(419) object file is not absolute (Objtohex)
The object file passed to OBJTOHEX has relocation items in it. This can indicate it is the wrong object file, or that the linker or OBJTOHEX have been given invalid options. The object output files from the assembler are relocatable, not absolute. The object file output of the linker is absolute.

(420) too many relocation items (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(421) too many segments (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(422) no end record (Linker)
This object file has no end record. This probably means it is not an object file. Contact Microchip Technical Support if the object file was generated by the compiler.

(423) illegal record type (Linker)
There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Contact Microchip Technical Support with details if the object file was created by the compiler.

(424) record too long (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(425) incomplete record (Objtohex, Libr)
The object file passed to OBJTOHEX or the librarian is corrupted. Contact Microchip Technical Support with details.

(427) syntax error in list (Objtohex)
There is a syntax error in a list read by OBJTOHEX. The list is read from standard input in response to an option.

(428) too many segment fixups (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(429) bad segment fixups (Objtohex)
This is an internal compiler error. Contact Microchip Technical Support with details.

(430) bad specification (Objtohex)
A list supplied to OBJTOHEX is syntactically incorrect.

(431) bad argument to -E (Objtoexe)
This option requires an integer argument in either base 8, 10, or 16. If you are invoking `objtoexe` directly then check this argument. Otherwise, this can be an internal compiler error and you should contact Microchip Technical Support with details.

(432) usage: objtohex [-ssymfile] [object-file [exe-file]] (Objtohex)
Improper usage of the command-line tool `objtohex`. If you are not invoking this tool directly, this is an internal compiler error and you should contact Microchip Technical Support with details.

(434) too many symbols (*) (Linker)
There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

(435) bad segment selector “*” (Linker)
The segment specification option (`-G`) to the linker is invalid, for example:

```
-GA/f0+10
```

Did you forget the radix?

```
-GA/f0h+10
```

(436) psect “*” re-orged (Linker)
This psect has had its start address specified more than once.

(437) missing “=” in class spec (Linker)
A class spec needs an `=` sign, e.g., `-Ctext=ROM`.

(438) bad size in -S option (Linker)
The address given in a `-S` specification is invalid, it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, for example:

```
-SCODE=f000
```

Did you forget the radix?

```
-SCODE=f000h
```

(439) bad -D spec: “*” (Linker)
The format of a `-D` specification, giving a delta value to a class, is invalid, for example:

```
-DCODE
```

What is the delta value for this class? Possibly, you meant something like:

```
-DCODE=2
```

(440) bad delta value in -D spec (Linker)
The delta value supplied to a `-D` specification is invalid. This value should an integer of base 8, 10, or 16.

(441) bad -A spec: “*” (Linker)
The format of a `-A` specification, giving address ranges to the linker, is invalid, for example:

```
-ACODE
```

What is the range for this class? Possibly, you meant:

```
-ACODE=0h-1fffh
```

(442) missing address in -A spec (Linker)
The format of a `-A` specification, giving address ranges to the linker, is invalid, for example:

```
-ACODE=
```

What is the range for this class? Possibly, you meant:

```
-ACODE=0h-1fffh
```

(443) bad low address "*" in -A spec (Linker)
The low address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, for example:

```
-ACODE=1fff-3fffh
```

Did you forget the radix?

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec (Linker)
There should be a minus sign, `-`, between the high and low addresses in a `-A` linker option, for example:

```
-AROM=1000h
```

Possibly, you meant:

```
-AROM=1000h-1fffh
```

(445) bad high address "*" in -A spec (Linker)
The high address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, for example:

```
-ACODE=0h-ffff
```

Did you forget the radix?

```
-ACODE=0h-ffffh
```

(446) bad overrun address "*" in -A spec (Linker)
The overrun address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, for example:

```
-AENTRY=0-0FFh-1FF
```

Did you forget the radix?

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address "*" in -A spec (Linker)
The load address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal, or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for HEX. A leading `0x` can also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, for example:

```
-ACODE=0h-3fffh/a000
```

Did you forget the radix?

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count "*" in -A spec (Linker)
The repeat count given in a `-A` specification is invalid, for example:

```
-AENTRY=0-0FFhxf
```

Did you forget the radix?

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec: * (Linker)
The `-A` spec is invalid. A valid `-A` spec should be something like:

```
-AROM=1000h-1FFFh
```

(450) psect "*" was never defined, or is local (Linker)
This psect has been listed in a `-P` option, but is not defined in any module within the program. Alternatively, the psect is defined using the `local` psect flag, but with no `class` flag; and, so, cannot be linked to an address. Check the assembly list file to ensure that the psect exists and that it is does not specify the local psect flag.

(451) bad psect origin format in -P option (Linker)
The origin format in a `-p` option is not a validly formed decimal, octal, or HEX number, nor is it the name of an existing psect. A HEX number must have a trailing `H`, for example:

```
-pbss=f000
```

Did you forget the radix?

```
-pbss=f000h
```

(452) bad "+" (minimum address) format in -P option (Linker)
The minimum address specification in the linker's `-p` option is badly formatted, for example:

```
-pbss=data+f000
```

Did you forget the radix?

```
-pbss=data+f000h
```

(453) missing number after "%" in -P option (Linker)
The `%` operator in a `-p` option (for rounding boundaries) must have a number after it.

(454) link and load address can't both be set to "." in -P option (Linker)
The link and load address of a psect have both been specified with a dot character. Only one of these addresses can be specified in this manner, for example:

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

Both of these options are valid and equivalent. However, the following usage is ambiguous:

```
-Pmypsect=./.
```

What is the link or load address of this psect?

(455) psect "*" not relocated on 0x* byte boundary (Linker)
This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the `-p` option. if necessary.

(456) psect "*" not loaded on 0x* boundary (Linker)
This psect has a relocatability requirement that is not met by the load address given in a `-p` option. For example, if a psect must be on a 4K byte boundary, you could not start it at 100H.

(459) remove failed; error: *, * (Xstrip)
The creation of the output file failed when removing an intermediate file.

(460) rename failed; error: *, * (Xstrip)
The creation of the output file failed when renaming an intermediate file.

(461) can't create * file "*" (Assembler, Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(464) missing key in avmap file (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(465) undefined symbol "*" in FNBREAK record (Linker)
The linker has found an undefined symbol in the `FNBREAK` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(466) undefined symbol "*" in FNINDIR record (Linker)
The linker has found an undefined symbol in the `FNINDIR` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(467) undefined symbol "*" in FNADDR record (Linker)
The linker has found an undefined symbol in the `FNADDR` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(468) undefined symbol "*" in FNCALL record (Linker)
The linker has found an undefined symbol in the `FNCALL` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(469) undefined symbol "*" in FNROOT record (Linker)
The linker has found an undefined symbol in the `FNROOT` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(470) undefined symbol "*" in FNSIZE record (Linker)
The linker has found an undefined symbol in the `FNSIZE` record for a non-reentrant function. Contact Microchip Technical Support if this is not handwritten assembler code.

(471) recursive function calls: (Linker)
These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the `reentrant` keyword (if supported with this compiler) or recode to avoid recursion, for example:

```
int test(int a)
{
    if(a == 5) {
    /* recursion cannot be supported by some compilers */
        return test(a++);
     }
    return 0;
}
```

(472) non-reentrant function “*” appears in multiple call graphs: rooted at “*” and “*” (Linker)
This function can be called from both main-line code and interrupt code. Use the `reentrant` keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function, for example:

```
void interrupt my_isr(void)
{
    scan(6);     /* scan is called from an interrupt function */
}
void process(int a)
{
    scan(a);     /* scan is also called from main-line code */
}
```

(473) function “*” is not called from specified interrupt_level (Linker)
The indicated function is never called from an interrupt function of the same interrupt level, for example:

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```

(474) no psect specified for function variable/argument allocation (Linker)
The `FNCONF` assembler directive which specifies to the linker information regarding the auto/parameter block was never seen. This is supplied in the standard runtime files if necessary. This error can imply that the correct run-time startup module was not linked. Ensure you have used the `FNCONF` directive if the runtime startup module is hand-written.

(475) conflicting FNCONF records (Linker)
The linker has seen two conflicting `FNCONF` directives. This directive should be specified only once and is included in the standard runtime startup code which is normally linked into every program.

(476) fixup overflow referencing * * (location 0x* (0x*+*), size *, value 0x*) (Linker)
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (1356) for more information.

(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) (Linker)
The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (1356) for more information.

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(479) circular indirect definition of symbol “*” (Linker)
The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

(480) function signatures do not match: * (*): 0x*/0x* (Linker)
The specified function has different signatures in different modules. This means it has been declared differently; i.e., it can have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible, for example:

```
extern int get_value(int in);
/* and in another module: */
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol "*" psect conflict (Linker)
A common symbol has been defined to be in more than one psect.

(482) symbol "*" is defined more than once in "*" (Assembler)
This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, for example:

```
_next:
    MOVE r0, #55
    MOVE [r1], r0
_next:          ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an underscore prepended to their name after compilation.

(483) symbol "*" can't be global (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(484) psect "*" can't be in classes "*" and "*" (Linker)
A psect cannot be in more than one class. This is either due to assembler modules with conflicting `class=` options to the `PSECT` directive, or use of the `-C` option to the linker, for example:

```
psect final,class=CODE
finish:
    /* elsewhere: */
psect final,class=ENTRY
```

(485) unknown "with" psect referenced by psect "*" (Linker)
The specified psect has been placed with a psect using the psect `with` flag. The psect it has been placed with does not exist, for example:

```
psect starttext,class=CODE,with=rext
; was that meant to be with text?
```

(486) psect "*" selector value redefined (Linker)
The `selector` value for this psect has been defined more than once.

(487) psect "*" type redefined: */* (Linker)
This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, i.e., linking 386 flat model code with 8086 real mode code.

(488) psect “*” memory space redefined: */* (Linker)
A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the space psect flag, for example:

```
psect spdata,class=RAM,space=0
    ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect “*” memory delta redefined: */* (Linker)
A `global` psect has been defined with two different `delta` values, for example:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class “*” memory space redefined: */* (Linker)
A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

(491) can’t find 0x* words for psect “*” in segment “*” (Linker)
One of the main tasks the linker performs is positioning the blocks (or psects) of code and data that is generated from the program into the memory available for the target device. This error indicates that the linker was unable to find an area of free memory large enough to accommodate one of the psects. The error message indicates the name of the psect that the linker was attempting to position and the segment name which is typically the name of a class which is defined by a `-A` linker option issued by the compiler driver. Typically psect names which are, or include, `text` relate to program code. Names such as `bss` or `data` refer to variable blocks. This error can be due to two reasons.

First, the size of the program or the program’s data has exceeded the total amount of space on the selected device. In other words, some part of your device’s memory has completely filled. If this is the case, then the size of the specified psect must be reduced.

The second cause of this message is when the total amount of memory needed by the psect being positioned is sufficient, but that this memory is fragmented in such a way that the largest contiguous block is too small to accommodate the psect. The linker is unable to split psects in this situation. That is, the linker cannot place part of a psect at one location and part somewhere else. Thus, the linker must be able to find a contiguous block of memory large enough for every psect. If this is the cause of the error, then the psect must be split into smaller psects if possible.

To find out what memory is still available, generate and look in the map file. Search for the string **UNUSED ADDRESS RANGES**. Under this heading, look for the name of the segment specified in the error message. If the name is not present, then all the memory available for this psect has been allocated. If it is present, there will be one address range specified under this segment for each free block of memory. Determine the size of each block and compare this with the number of words specified in the error message.

Psects containing code can be reduced by using all the compiler’s optimizations, or restructuring the program. If a code psect must be split into two or more small psects, this requires splitting a function into two or more smaller functions (which can call each other). These functions can need to be placed in new modules.

Psects containing data can be reduced when invoking the compiler optimizations, but the effect is less dramatic. The program can need to be rewritten so that it needs less variables. If the default linker options must be changed, this can be done indirectly through the driver using the driver `-Wl,` option. If a data psect cannot be positioned, then you typically need to reduce the total size of variables being used.

For example, after receiving the message:

```
Can’t find 0x34 words (0x34 withtotal) for psect text in segment CODE (error)
```

look in the map file for the ranges of unused memory.

```
UNUSED ADDRESS RANGES
CODE             00000244-0000025F
                 00001000-0000102f
RAM              00300014-00301FFB
```

In the `CODE` segment, there is 0x1c (0x25f-0x244+1) bytes of space available in one block and 0x30 available in another block. Neither of these are large enough to accommodate the psect `text` which is 0x34 bytes long. Notice that the total amount of memory available is larger than 0x34 bytes. If the function that is encoded into the `text` psect can be split into two smaller functions, there is a chance the program will link correctly.

(492) attempt to position absolute psect "*" is illegal (Linker)
This psect is absolute and should not have an address specified in a `-P` option. Either remove the `abs` psect flag, or remove the `-P` linker option.

(493) origin of psect "*" is defined more than once (Linker)
The origin of this psect is defined more than once. There is most likely more than one `-p` linker option specifying this psect.

(494) bad -P format "*/*" (Linker)
The `-P` option given to the linker is malformed. This option specifies placement of a psect, for example:

```
-Ptext=10g0h
```

Possibly, you meant:

```
-Ptext=10f0h
```

(495) use of both "with=" and "INCLASS/INCLASS" allocation is illegal (Linker)
It is not legal to specify both the link and location of a psect as within a class, when that psect was also defined using a with psect flag.

(497) psect "*" exceeds max size: *h > *h (Linker)
The psect has more bytes in it than the maximum allowed as specified using the `size` psect flag.

(498) psect "*" exceeds address limit: *h > *h (Linker)
The maximum address of the psect exceeds the limit placed on it using the `limit` psect flag. Either the psect needs to be linked at a different location or there is too much code/data in the psect.

(499) undefined symbol: (Assembler, Linker)
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

8.2 Messages 500 Thru 999

(500) undefined symbols: (Linker)
A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(501) program entry point is defined more than once (Linker)
There is more than one entry point defined in the object files given the linker. End entry point is specified after the `END` directive. The runtime startup code defines the entry point, for example:

```
powerup:
goto start
END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * (Linker)
An object file contained a record with an illegal size. This probably means the file is truncated or not an object file. Contact Microchip Technical Support with details.

(503) ident records do not match (Linker)
The object files passed to the linker do not have matching `ident` records. This means they are for different device types.

(504) object code version is greater than *.* (Linker)
The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker. Contact Microchip Technical Support if you have not patched the linker.

(505) no end record found inobject file (Linker)
An object file did not contain an end record. This probably means the file is corrupted or not an object file. Contact Microchip Technical Support if the object file was generated by the compiler.

(506) object file record too long: *+* (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(507) unexpected end of file in object file (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(508) relocation offset (*) out of range 0..*-*-1 (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(509) illegal relocation size: * (Linker)
There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker. Contact Microchip Technical Support with details if the object file was created by the compiler.

(510) complex relocation not supported for -R or -L options (Linker)
The linker was given a `-R` or `-L` option with file that contain complex relocation.

(511) bad complex range check (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(512) unknown complex operator 0x* (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(513) bad complex relocation (Linker)
The linker has been asked to perform complex relocation that is not syntactically correct. Probably means an object file is corrupted.

(514) illegal relocation type: * (Linker)
An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file. Contact Microchip Technical Support with details if the object file was created by the compiler.

(515) unknown symbol type * (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(516) text record has bad length: *-*-(*+1) < 0 (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(520) function “*” is never called (Linker)
This function is never called. This cannot represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code. Some assembler library routines are never called, although they are actually execute. In this case, the routines are linked in a special sequence so that program execution falls through from one routine to the next.

(521) call depth exceeded by function “*” (Linker)
The call graph shows that functions are nested to a depth greater than specified.

(522) library “*” is badly ordered (Linker)
This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

(523) argument to -W option (*) illegal and ignored (Linker)
The argument to the linker option `-w` is out of range. This option controls two features. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

(524) unable to open list file “*”: * (Linker)
The named list file could not be opened. The linker would be trying to fixup the list file so that it will contain absolute addresses. Ensure that an assembler list file was generated during the compilation stage. Alternatively, remove the assembler list file generation option from the link step.

(525) too many address (memory) spaces; space (*) ignored (Linker)
The limit to the number of address spaces (specified with the `PSECT` assembler directive) is currently 16.

(526) psect “*” not specified in -P option (first appears in “*”) (Linker)
This psect was not specified in a `-P` or `-A` option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

(528) no start record; entry point defaults to zero (Linker)
None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This can be harmless, but it is recommended that you define a start address in your startup module by using the `END` directive.

(529) usage: objtohex [-Ssymfile] [object-file [HEX-file]] (Objtohex)
Improper usage of the command-line tool `objtohex`. If you are not invoking this tool directly, this is an internal compiler error, and you should contact Microchip Technical Support with details.

(593) can’t find 0x* words (0x* withtotal) for psect “*” in segment “*” (Linker)
See message (491).

(594) undefined symbol: (Linker)
The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(595) undefined symbols: (Linker)
A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(596) segment “*” (*-*) overlaps segment “*” (*-*) (Linker)
The named segments have overlapping code or data. Check the addresses being assigned by the `-P` linker option.

(599) No psect classes given for COFF write (Cromwell)
CROMWELL requires that the program memory psect classes be specified to produce a COFF file. Ensure that you are using the `-N` option.

(600) No chip arch given for COFF write (Cromwell)
CROMWELL requires that the chip architecture be specified to produce a COFF file. Ensure that you are using the `-P` option.

(601) Unknown chip arch “*” for COFF write (Cromwell)
The chip architecture specified for producing a COFF file isn’t recognized by CROMWELL. Ensure that you are using the `-P` option, and that the architecture is correctly specified.

(602) null file format name (Cromwell)
The `-I` or `-O` option to CROMWELL must specify a file format.

(603) ambiguous file format name “*” (Cromwell)
The input or output format specified to CROMWELL is ambiguous. These formats are specified with the `-i` key and `-o` key options respectively.

(604) unknown file format name “*” (Cromwell)
The output format specified to CROMWELL is unknown, for example:

```
cromwell -m -P16F877 main.HEX main.sym -ocot
```

There is no output file type of cot. Did you mean cof?

(605) did not recognize format of input file (Cromwell)
The input file to CROMWELL is required to have a Cromwell map file (CMF), COD, Intel HEX, Motorola HEX, COFF, OMF51, ELF, UBROF or HI-TECH format.

(606) inconsistent symbol tables (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(607) inconsistent line number tables (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(608) bad path specification (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(609) missing device spec after -P (Cromwell)
The `-p` option to CROMWELL must specify a device name.

(610) missing psect classes after -N (Cromwell)
CROMWELL requires that the `-N` option be given a list of the names of psect classes.

(611) too many input files (Cromwell)
To many input files have been specified to be converted by CROMWELL.

(612) too many output files (Cromwell)
To many output file formats have been specified to CROMWELL.

(613) no output file format specified (Cromwell)
The output format must be specified to CROMWELL.

(614) no input files specified (Cromwell)
CROMWELL must have an input file to convert.

(616) option -Cbaseaddr is illegal with options -R or -L (Linker)
The linker option `-Cbaseaddr` cannot be used in conjunction with either the `-R` or `-L` linker options.

(618) error reading COD file data (Cromwell)
An error occurred reading the input COD file. Confirm the spelling and path of the file specified on the command line.

(619) I/O error reading symbol table (Cromwell)
The COD file has an invalid format in the specified record.

(620) filename index out of range in line number record (Cromwell)
The COD file has an invalid value in the specified record.

(621) error writing ELF/DWARF section “*” on “*” (Cromwell)
An error occurred writing the indicated section to the given file. Confirm the spelling and path of the file specified on the command line.

(622) too many type entries (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(623) bad class in type hashing (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(624) bad class in type compare (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(625) too many files in COFF file (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(626) string lookup failed in COFF: get_string() (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(627) missing “*” in SDB file “*” line * column * (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(629) bad storage class “*” in SDB file “*” line * column * (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(630) invalid syntax for prefix list in SDB file “*” (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(631) syntax error at token "*" in SDB file "*" line * column * (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(632) can't handle address size (*) (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(633) unknown symbol class (*) (Cromwell)
CROMWELL has encountered a symbol class in the symbol table of a COFF, Microchip COFF, or ICOFF file which it cannot identify.

(634) error dumping "*" (Cromwell)
Either the input file to CROMWELL is of an unsupported type or that file cannot be dumped to the screen.

(635) invalid HEX file "*" on line * (Cromwell)
The specified HEX file contains an invalid line. Contact Microchip Technical Support if the HEX file was generated by the compiler.

(636) error in Intel HEX file "*" on line * (Cromwell, Hexmate)
An error was found at the specified line in the specified Intel HEX file. The HEX file may be corrupt.

(637) unknown prefix "*" in SDB file "*" (Cromwell)
This is an internal compiler warning. Contact Microchip Technical Support with details.

(638) version mismatch: 0x* expected (Cromwell)
The input Microchip COFF file wasn't produced using CROMWELL.

(639) zero bit width in Microchip optional header (Cromwell)
The optional header in the input Microchip COFF file indicates that the program or data memory spaces are zero bits wide.

(668) prefix list did not match any SDB types (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(669) prefix list matched more than one SDB type (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

(670) bad argument to -T (Clist)
The argument to the `-T` option to specify tab size was not present or correctly formed. The option expects a decimal integer argument.

(671) argument to -T should be in range 1 to 64 (Clist)
The argument to the `-T` option to specify tab size was not in the expected range. The option expects a decimal integer argument ranging from 1 to 64 inclusive.

(673) missing filename after * option (Objtohex)
The indicated option requires a valid file name. Ensure that the filename argument supplied to this option exists and is spelled correctly.

(674) too many references to "*" (Cref)
This is an internal compiler error. Contact Microchip Technical Support with details.

(679) unknown extraspecial: * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(680) bad format for -P option (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(681) bad common spec in -P option (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(685) bad putwsize() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(686) bad switch size (*) (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(687) bad pushreg “*” (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(688) bad popreg “*” (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(689) unknown predicate “*” (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(691) interrupt functions not implemented for 12 bit PIC MCU (Code Generator)
The 12-bit (Baseline) range of PIC MCU processors do not support interrupts.

(692) more than one interrupt level is associated with the interrupt function “*” (Code Generator)
Only one interrupt level can be associated with an interrupt function. Check to ensure that only one `interrupt_level` pragma has been used with the function specified. This pragma can be used more than once on main-line functions that are called from interrupt functions. For example:

```
#pragma interrupt_level 0
#pragma interrupt_level 1  /* oops -- which is it to be: 0 or 1? */
void interrupt isr(void)
{
```

(693) 0 (default) or 1 are the only acceptable interrupt levels for this function (Code Generator)
The only possible interrupt levels are 0 or 1. Check to ensure that all `interrupt_level` pragmas use these levels.

```
#pragma interrupt_level 2  /* oops -- only 0 or 1 */
void interrupt isr(void)
{
    /* isr code goes here */
}
```

(694) no interrupt strategy available (Code Generator)
The device does not support saving and subsequent restoring of registers during an interrupt service routine.

(695) duplicate case label (*) (Code Generator)
There are two `case` labels with the same value in this `switch` statement, for example:

```
switch(in) {
case '0':  /* if this is case '0'... */
    b++;
    break;
case '0':  /* then what is this case? */
    b--;
```

```
    break;
}
```

(696) out-of-range case label (*) (Code Generator)
This case label is not a value that the controlling expression can yield, thus this label will never be selected.

(697) non-constant case label (Code Generator)
A `case` label in this switch statement has a value which is not a constant.

(698) bit variables must be global or static (Code Generator)
A `__bit` variable cannot be of type auto. If you require a bit variable with scope local to a block of code or function, qualify it static, for example:

```
bit proc(int a)
{
    bit bb;         /* oops --  this should be: static bit bb; */
    bb = (a > 66);
    return bb;
}
```

(699) no case labels in switch (Code Generator)
There are no `case` labels in this `switch` statement, for example:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(700) truncation of enumerated value (Code Generator)
An enumerated value larger than the maximum value supported by this compiler was detected and has been truncated, for example:

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(701) unreasonable matching depth (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(702) regused(): bad arg to G (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(703) bad GN (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(704) bad RET_MASK (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(705) bad which (*) after I (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(706) bad which in expand() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(707) bad SX (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(708) bad mod “+” for how = “*” (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(709) metaregister “*” can’t be used directly (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(710) bad U usage (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(711) bad how in expand() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(712) can’t generate code for this expression (Code Generator)
This error indicates that a C expression is too difficult for the code generator to actually compile. For successful code generation, the code generator must know how to compile an expression and there must be enough resources (i.e., registers or temporary memory locations) available. Simplifying the expression, i.e., using a temporary variable to hold an intermediate result, can often bypass this situation.

This error can also be issued if the code being compiled is unusual. For example, code which writes to a `const`-qualified object is illegal and will result in warning messages, but the code generator can unsuccessfully try to produce code to perform the write.

This error can also result from an attempt to redefine a function that uses the `intrinsic` pragma.

(713) bad initialization list (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(714) bad intermediate code (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(715) bad pragma “*” (Code Generator)
The code generator has been passed a `pragma` directive that it does not understand. This implies that the `pragma` you have used is not implemented for the target device.

(716) bad argument to -M option “*” (Code Generator)
The code generator has been passed a `-M` option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

(718) incompatible intermediate code version; should be *.* (Code Generator)
The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the `TEMP` environment variable. If it refers to a long path name, change it to something shorter. Contact Microchip Technical Support with details if required.

(720) multiple free: * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(721) element count must be constant expression (Code Generator)
The expression that determines the number of elements in an array must be a constant expression. Variables qualified as `const` do not form such an expression.

```
const unsigned char mCount = 5;
int mDeadtimeArr[mCount];  // oops -- the size cannot be a variable
```

(722) bad variable syntax in intermediate code (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(723) function definitions nested too deep (Code Generator)
This error is unlikely to happen with C code, because C cannot have nested functions! Contact Microchip Technical Support with details.

(724) bad op (*) in revlog() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(726) bad op “*” in uconval() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(727) bad op “*” in bconfloat() (Code Generator)
This is an internal code generator error. Contact Microchip Technical Support with details.

(728) bad op “*” in confloat() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(729) bad op “*” in conval() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(730) bad op “*” (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(731) expression error with reserved word (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(732) initialization of bit types is illegal (Code Generator)
Variables of type `__bit` cannot be initialized, for example:

```
__bit b1 = 1; /* oops!  b1 must be assigned after its definition */
```

(733) bad string “*” in pragma “psect” (Code Generator)
The code generator has been passed a `#pragma psect` directive that has a badly formed string, for example:

```
#pragma psect text  /* redirect text psect into what? */
```

Possibly, you meant something like:

```
#pragma psect text=special_text
```

(734) too many “psect” pragmas (Code Generator)
Too many `#pragma psect` directives have been used.

(735) bad string “*” in pragma “stack_size” (Code Generator)
The argument to the `stack_size` pragma is malformed. This pragma must be followed by a number representing the maximum allowed stack size.

(737) unknown argument “*” to pragma “switch” (Code Generator)
The `#pragma` switch directive has been used with an invalid `switch` code generation method. Possible arguments are: `auto`, `simple` and `direct`.

(739) error closing output file (Code Generator)
The compiler detected an error when closing a file. Contact Microchip Technical Support with details.

(740) zero dimension array is illegal (Code Generator)
The code generator has been passed a declaration that results in an array having a zero dimension.

(741) bitfield too large (* bits) (Code Generator)
The maximum number of bits in a bit-field is 8, the same size as the storage unit width.

```
struct {
    unsigned flag  : 1;
    unsigned value : 12;  /* oops -- that's larger than 8 bits wide */
    unsigned cont  : 6;
} object;
```

(742) function "*" argument evaluation overlapped (Linker)
A function call involves arguments which overlap between two functions. This could occur with a call like:

```
void fn1(void)
{
    fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
    return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
    return one+two+three;
}
```

where `fn1` is calling `fn3` , and two arguments are evaluated by calling `fn2` , which in turn calls `fn3`. The program structure should be modified to prevent this type of call sequence.

(743) divide by zero (Code Generator)
An expression involving a division by zero has been detected in your code.

(744) static object "*" has zero size (Code Generator)
A `static` object has been declared, but has a size of zero.

(745) nodecount = * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(746) object "*" qualified const but not initialized (Code Generator)
An object has been qualified as `const`, but there is no initial value supplied at the definition. As this object cannot be written by the C program, this can imply the initial value was accidentally omitted.

(747) unrecognized option "*" to -Z (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(748) variable "*" possibly used before being assigned a value (Code Generator)
This variable has possibly been used before it was assigned a value. Because it is an auto variable, this will result in it having an unpredictable value, for example:

```
int main(void)
{
    int a;
    if(a)         /* oops -- 'a' has never been assigned a value */
```

```
        process();
}
```

(749) unknown register name “*” used with pragma (Linker)
This is an internal compiler error. Contact Microchip Technical Support with details.

(750) constant operand to || or && (Code Generator)
One operand to the logical operators `||` or `&&` is a constant. Check the expression for missing or badly placed parentheses. This message can also occur if the global optimizer is enabled and one of the operands is an auto or static local variable whose value has been tracked by the code generator, for example:

```
{
    int a;
    a = 6;
    if(a || b)  /* a is 6, therefore this is always true */
        b++;
```

(751) arithmetic overflow in constant expression (Code Generator)
A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression. The most common code to trigger this warning is assignments to signed data types. For example:

```
signed char c;
c = 0xFF;
```

As a signed 8-bit quantity, `c` can only be assigned values -128 to 127. The constant is equal to 255 and is outside this range. If you mean to set all bits in this variable, then use either of:

```
c = ~0x0;
c = -1;
```

which sets all the bits in the variable, regardless of variable size and without warning.

This warning can also be triggered by intermediate values overflowing. For example:

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

A quick check with your calculator reveals that 240 * 137 is 32880 which can easily be stored in an `unsigned int`, but a warning is produced. Why? Because 240 and 137 and both `signed int` values. Therefore the result of the multiplication must also be a `signed int` value, but a `signed int` cannot hold the value 32880. Both operands are constant values so the code generator can evaluate this expression at compile time, but it must do so following all the ANSI C rules. The following code forces the multiplication to be performed with an unsigned result:

```
i = 240u * 137;  /* force at least one operand
to be unsigned */
```

(752) conversion to shorter data type (Code Generator)
Truncation can occur in this expression as the lvalue is of shorter type than the rvalue, for example:

```
char a;
int b, c;
a = b + c;  /* int to char conversion can result in truncation */
```

(753) undefined shift (* bits) (Code Generator)
An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard, for example:

```
int input;
input <<= 33;  /* oops -- that shifts the entire value out */
```

(754) bitfield comparison out of range (Code Generator)
This is the result of comparing a bit-field with a value when the value is out of range of the bit-field. That is, comparing a 2-bit bit-field to the value 5 will never be true as a 2-bit bit-field has a range from 0 to 3. For example:

```
struct {
    unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
    return (value.mask == 6);  /* test can
}
```

(755) divide by zero (Code Generator)
A constant expression that was being evaluated involved a division by zero, for example:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch (Code Generator)
A conditional branch (generated by an `if`, `for`, `while` statement etc.) always follows the same path. This will be some sort of comparison involving a variable and a constant expression. For the code generator to issue this message, the variable must have local scope (either auto or static local) and the global optimizer must be enabled, possibly at higher level than 1, and the warning level threshold can need to be lower than the default level of 0.

The global optimizer keeps track of the contents of local variables for as long as is possible during a function. For C code that compares these variables to constants, the result of the comparison can be deduced at compile time and the output code hard coded to avoid the comparison, for example:

```
{
int a, b;
a = 5;
/* this can never be false; always perform the true statement */
if(a == 5)
    b = 6;
```

will produce code that sets a to 5, then immediately sets b to 6.

No code will be produced for the comparison `if(a == 5)`. If a was a global variable, it can be that other functions (particularly interrupt functions) can modify it and so tracking the variable cannot be performed.

This warning can indicate more than an optimization made by the compiler. It can indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected.

This warning can also be issued because you have written something like `while(1)`. To produce an infinite loop, use `for(;;)`.

A similar situation arises with for loops, for example:

```
{
int a, b;
/* this loop must iterate at least once */
for(a=0; a!=10; a++)
    b = func(a);
```

In this case the code generator can again pick up that a is assigned the value 0, then immediately checked to see if it is equal to 10. Because a is modified during the for loop, the comparison code cannot be removed, but the code generator will adjust the code so that the comparison is not performed on the first pass of the loop; only on the subsequent passes. This cannot reduce code size, but it will speed program execution.

(758) constant conditional branch: possible use of “=” instead of “==” (Code Generator)
There is an expression inside an `if` or other conditional construct, where a constant is being assigned to a variable. This can mean you have inadvertently used an assignment = instead of a compare ==, for example:

```
int a, b;
/* this can never be false; always perform the true statement */
if(a = 4)
    b = 6;
```

will assign the value 4 to `a`, then , as the value of the assignment is always true, the comparison can be omitted and the assignment to b always made. Did you mean:

```
/* this can never be false;
always perform the true statement */
if(a == 4)
    b = 6;
```

which checks to see if a is equal to 4.

(759) expression generates no code (Code Generator)
This expression generates no output code. Check for things like leaving off the parentheses in a function call, for example:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

Some devices require that special function register need to be read to clear hardware flags. To accommodate this, in some instances the code generator does produce code for a statement which only consists of a variable ID. This can happen for variables which are qualified as volatile. Typically the output code will read the variable, but not do anything with the value read.

(760) portion of expression has no effect (Code Generator)
Part of this expression has no side effects and no effect on the value of the expression, for example:

```
int a, b, c;
a = b,c;  /* "b" has no effect, was that meant to be a comma? */
```

(761) size of yields 0 (Code Generator)
The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer; i.e., you can have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

(762) constant truncated when assigned to bitfield (Code Generator)
A constant value is too large for a bit-field structure member to which it is being assigned, for example:

```
struct INPUT {
    unsigned a : 3;
    unsigned b : 5;
} input_grp;
input_grp.a = 0x12;  /* oops -- 0x12 cannot fit into a 3-bit wide object */
```

(763) constant left operand to “? :” operator (Code Generator)
The left operand to a conditional operator `?` is constant, thus the result of the tertiary operator `? :` will always be the same, for example:

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison (Code Generator)
A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, for example:

```
unsigned char c;
if(c > 300)      /* oops -- how can this be true? */
    close();
```

(765) degenerate unsigned comparison (Code Generator)
There is a comparison of an unsigned value with zero, which will always be true or false, for example:

```
unsigned char c;
if(c >= 0)
    ...
```

will always be true, because an unsigned value can never be less than zero.

(766) degenerate signed comparison (Code Generator)
There is a comparison of a signed value with the most negative value possible for this type, such that the comparison will always be true or false, for example:

```
char c;
if(c >= -128)
    ...
```

will always be true, because an 8 bit signed char has a maximum negative value of -128.

(767) constant truncated to bitfield width (Code Generator)
A constant value is too large for a bit-field structure member on which it is operating, for example:

```
struct INPUT {
    unsigned a : 3;
    unsigned b : 5;
} input_grp;
input_grp.a |= 0x13;   /* oops -- 0x13 to large for 3-bit wide object */
```

(768) constant relational expression (Code Generator)
There is a relational expression that will always be true or false. This, for example, can be the result of comparing an unsigned number with a negative value; or comparing a variable with a value greater than the largest number it can represent, for example:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
b = 9;
```

(769) no space for macro definition (Assembler)
The assembler has run out of memory.

(772) include files nested too deep (Assembler)
Macro expansions and include file handling have filled up the assembler’s internal stack. The maximum number of open macros and include files is 30.

(773) macro expansions nested too deep (Assembler)
Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

(774) too many macro parameters (Assembler)
There are too many macro parameters on this macro definition.

(776) can't allocate space for object "*" (offs: *) (Assembler)
The assembler has run out of memory.

(777) can't allocate space for opnd structure within object "*" (offs: *) (Assembler)
The assembler has run out of memory.

(780) too many psects defined (Assembler)
There are too many psects defined! Boy, what a program!

(781) can't enter abs psect (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(782) REMSYM error (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(783) "with" psects are cyclic (Assembler)
If Psect A is to be placed "with" Psect B, and Psect B is to be placed "with" Psect A, there is no hierarchy. The `with` flag is an attribute of a psect and indicates that this psect must be placed in the same memory page as the specified psect.

Remove a with flag from one of the psect declarations. Such an assembler declaration can look like:

```
psect my_text,local,class=CODE,with=basecode
```

which will define a psect called `my_text` and place this in the same page as the psect basecode.

(784) overfreed (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(785) too many temporary labels (Assembler)
There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

(787) can't handle "v_rtype" of * in copyexpr (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(788) invalid character "*" in number (Assembler)
A number contained a character that was not part of the range 0-9 or 0-F.

(790) end of file inside conditional (Assembler)
END-of-FILE was encountered while scanning for an `endif` to match a previous `if`.

(793) unterminated macro argument (Assembler)
An argument to a macro is not terminated. Note that angle brackets, `< >`, are used to quote macro arguments.

(794) invalid number syntax (Assembler)
The syntax of a number is invalid. This, for example, can be use of 8 or 9 in an octal number, or other malformed numbers.

(796) use of LOCAL outside macros is illegal (Assembler)
The `LOCAL` directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

(797) syntax error in LOCAL argument (Assembler)
A symbol defined using the `LOCAL` assembler directive in an assembler macro is syntactically incorrect. Ensure that all symbols and all other assembler identifiers conform with the assembly language of the target device.

(798) use of macro arguments in a LOCAL directive is illegal (Assembler)
The list of labels after the directive `LOCAL` cannot include any of the formal parameters to an enclosing macro, for example:

```
mmm MACRO a1
MOVE        r0, #a1
LOCAL        a1    ; oops -- the parameter cannot be used with LOCAL
ENDM
```

(799) REPT argument must be >= 0 (Assembler)
The argument to a `REPT` directive must be greater than zero, for example:

```
REPT -2                         ; -2 copies of this code? */
MOVE            r0, [r1]++
ENDM
```

(800) undefined symbol “*” (Assembler)
The named symbol is not defined in this module and has not been specified `GLOBAL`.

(801) range check too complex (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(802) invalid address after END directive (Assembler)
The start address of the program which is specified after the assembler `END` directive must be a label in the current file.

(803) undefined temporary label (Assembler)
A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

(804) write error on object file (Assembler)
The assembler failed to write to an object file. This can be an internal compiler error. Contact Microchip Technical Support with details.

(806) attempted to get an undefined object (*) (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(807) attempted to set an undefined object (*) (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(808) bad size in add_reloc() (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(809) unknown addressing mode (*) (Assembler)
An unknown addressing mode was used in the assembly file.

(811) “cnt” too large (*) in display() (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(814) device type not defined (Assembler)
The device must be defined either from the command line (e.g., `-16c84`), via the device assembler directive, or via the `LIST` assembler directive.

(815) syntax error in chipinfo file at line * (Assembler)
The chipinfo file contains non-standard syntax at the specified line.

(816) duplicate ARCH specification in chipinfo file “*” at line * (Assembler, Driver)
The chipinfo file has a device section with multiple `ARCH` values. Only one `ARCH` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(817) unknown architecture in chipinfo file at line * (Assembler, Driver)
An chip architecture (family) that is unknown was encountered when reading the chip INI file.

(818) duplicate BANKS for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `BANKS` values. Only one `BANKS` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(819) duplicate ZEROREG for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `ZEROREG` values. Only one `ZEROREG` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(820) duplicate SPAREBIT for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `SPAREBIT` values. Only one `SPAREBIT` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(821) duplicate INTSAVE for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `INTSAVE` values. Only one `INTSAVE` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(822) duplicate ROMSIZE for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `ROMSIZE` values. Only one `ROMSIZE` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(823) duplicate START for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `START` values. Only one `START` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(824) duplicate LIB for “*” in chipinfo file at line * (Assembler)
The chipinfo file has a device section with multiple `LIB` values. Only one `LIB` value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(825) too many RAMBANK lines in chipinfo file for “*” (Assembler)
The chipinfo file contains a device section with too many `RAMBANK` fields. Reduce the number of values.

(826) inverted ram bank in chipinfo file at line * (Assembler, Driver)
The second HEX number specified in the `RAM` field in the chipinfo file must be greater in value than the first.

(827) too many COMMON lines in chipinfo file for "*" (Assembler)
There are too many lines specifying common (access bank) memory in the chip configuration file.

(828) inverted common bank in chipinfo file at line * (Assembler, Driver)
The second HEX number specified in the `COMMON` field in the chipinfo file must be greater in value than the first. Contact Microchip Technical Support if you have not modified the chipinfo INI file.

(829) unrecognized line in chipinfo file at line * (Assembler)
The chipinfo file contains a device section with an unrecognized line. Contact Microchip Technical Support if the INI has not been edited.

(830) missing ARCH specification for "*" in chipinfo file (Assembler)
The chipinfo file has a device section without an `ARCH` values. The architecture of the device must be specified. Contact Microchip Technical Support if the chipinfo file has not been modified.

(832) empty chip info file "*" (Assembler)
The chipinfo file contains no data. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(833) no valid entries in chipinfo file (Assembler)
The chipinfo file contains no valid device descriptions.

(834) page width must be >= 60 (Assembler)
The listing page width must be at least 60 characters. Any less will not allow a properly formatted listing to be produced, for example:

```
LIST C=10  ; the page width will need to be wider than this
```

(835) form length must be >= 15 (Assembler)
The form length specified using the `-F` length option must be at least 15 lines. Setting this length to zero is allowed and turns off paging altogether. The default value is zero (pageless).

(836) no file arguments (Assembler)
The assembler has been invoked without any file arguments. It cannot assemble anything.

(839) relocation too complex (Assembler)
The complex relocation in this expression is too big to be inserted into the object file.

(840) phase error (Assembler)
The assembler has calculated a different value for a symbol on two different passes. This is commonly due to the redefinition of a psect with conflicting delta values.

(841) bad source/destination for movfp/movpf instruction (Assembler)
The absolute address specified with the `movfp`/`movpf` instruction is too large.

(842) bad bit number (Assembler)
A bit number must be an absolute expression in the range 0-7.

(843) a macro name can't also be an EQU/SET symbol (Assembler)
An `EQU` or `SET` symbol has been found with the same name as a macro. This is not allowed. For example:

```
getval MACRO
MOV  r0, r1
```

```
ENDM
getval EQU 55h  ; oops -- choose a different name to the macro
```

(844) lexical error (Assembler)
An unrecognized character or token has been seen in the input.

(845) symbol “*” defined more than once (Assembler)
This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, for example:

```
_next:
MOVE        r0, #55
MOVE        [r1], r0
_next:           ; oops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an underscore prepended to their name after compilation.

(846) relocation error (Assembler)
It is not possible to add together two relocatable quantities. A constant can be added to a relocatable value and two relocatable addresses in the same psect can be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

(847) operand error (Assembler)
The operand to this opcode is invalid. Check your assembler reference manual for the proper form of operands for this instruction.

(848) label defined in this module has also been declared EXTRN (Assembler)
The definition for an assembly label and an `EXTRN` declaration for the same symbol, appear in the same module. Use `GLOBAL` instead of `EXTRN` if you want this symbol to be accessible from other modules.

(849) illegal instruction for this device (Assembler)
The instruction is not supported by this device.

(850) PAGESEL not usable with this device (Assembler)
The `PAGESEL` pseudo-instruction is not usable with the device selected.

(851) illegal destination (Assembler)
The destination (either `,f` or `,w`) is not correct for this instruction.

(852) radix must be from 2 - 16 (Assembler)
The radix specified using the `RADIX` assembler directive must be in the range from 2 (binary) to 16 (hexadecimal).

(853) invalid size for FNSIZE directive (Assembler)
The assembler `FNSIZE` assembler directive arguments must be positive constants.

(855) ORG argument must be a positive constant (Assembler)
An argument to the `ORG` assembler directive must be a positive constant or a symbol which has been equated to a positive constant, for example:

```
ORG -10  /* this must a positive offset to the current psect */
```

(856) ALIGN argument must be a positive constant (Assembler)
The `ALIGN` assembler directive requires a non-zero positive integer argument.

(857) use of both local and global psect flags is illegal with same psect (Linker)
A `local` psect cannot have the same name as a `global` psect, for example:

```
psect text,class=CODE       ; the text psect is implicitly global
MOVE        r0, r1
; elsewhere:
psect text,local,class=CODE
MOVE        r2, r4
```

The `global` flag is the default for a psect if its scope is not explicitly stated.

(859) argument to C option must specify a positive constant (Assembler)
The parameter to the `LIST` assembler control's `C=` option (which sets the column width of the listing output) must be a positive decimal constant number, for example:

```
LIST C=a0h  ; constant must be decimal and positive,
try: LIST C=80
```

(860) page width must be >= 49 (Assembler)
The page width suboption to the `LIST` assembler directive must specify a width of at least 49.

(861) argument to N option must specify a positive constant (Assembler)
The parameter to the `LIST` assembler control's `N` option (which sets the page length for the listing output) must be a positive constant number, for example:

```
LIST N=-3  ; page length must be positive
```

(862) symbol is not external (Assembler)
A symbol has been declared as `EXTRN` but is also defined in the current module.

(863) symbol can't be both extern and public (Assembler)
If the symbol is declared as `extern`, it is to be imported. If it is declared as `public`, it is to be exported from the current module. It is not possible for a symbol to be both.

(864) argument to "size" psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive's `size` flag must be a positive constant number, for example:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect flag "size" redefined (Assembler)
The `size` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, for example:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) argument to "reloc" psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive's `reloc` flag must be a positive constant number, for example:

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect flag “reloc” redefined (Assembler)
The `reloc` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, for example:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) argument to “delta” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `DELTA` flag must be a positive constant number, for example:

```
PSECT text,class=CODE,delta=-2  ; negative delta value doesn’t make sense
```

(869) psect flag “delta” redefined (Assembler)
The `DELTA` option of a psect has been redefined more than once in the same module.

(870) argument to “pad” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `PAD` flag must be a non-zero positive integer.

(871) argument to “space” psect flag must specify a positive constant (Assembler)
The parameter to the `PSECT` assembler directive’s `space` flag must be a positive constant number, for example:

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect flag “space” redefined (Assembler)
The `space` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, for example:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) a psect can only be in one class (Assembler)
You cannot assign a psect to more than one class. The psect was defined differently at this point than when it was defined elsewhere. A psect’s class is specified via a flag as in the following:

```
psect text,class=CODE
```

Look for other psect definitions that specify a different class name.

(874) a psect can only have one “with” option (Assembler)
A psect can only be placed with one other psect. Look for other psect definitions that specify a different with psect name. A psect’s with option is specified via a flag, as shown in the following:

```
psect bss,with=data
; elsewhere
psect bss,with=lktab  ; oops -- bss is to be linked with two psects
```

(875) bad character constant in expression (Assembler)
The character constant was expected to consist of only one character, but was found to be greater than one character or none at all. An assembler specific example:

```
MOV  r0, #’12’    ; ’12’ specifies two characters
```

(876) syntax error (Assembler)
A syntax error has been detected. This could be caused a number of things.

(877) yacc stack overflow (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(878) -S option used: "*" ignored (Driver)
The indicated assembly file has been supplied to the driver in conjunction with the `-S` option. The driver really has nothing to do because the file is already an assembly file.

(880) invalid number of parameters. Use "* –HELP" for help (Driver)
Improper command-line usage of the of the compiler's driver.

(881) setup succeeded (Driver)
The compiler has been successfully setup using the `--setup` driver option.

(883) setup failed (Driver)
The compiler was not successfully setup using the `--setup` driver option. Ensure that the directory argument to this option is spelled correctly, is syntactically correct for your host operating system and it exists.

(884) please ensure you have write permissions to the configuration file (Driver)
The compiler was not successfully setup using the `--setup` driver option because the driver was unable to access the XML configuration file. Ensure that you have write permission to this file. The driver will search the following configuration files in order:

- the file specified by the environment variable XC_XML
- the file /etc/xc.xml if the directory '/etc ' is writable and there is no .xc.xml file in your home directory
- the file .xc.xml file in your home directory

If none of the files can be located, then the above error will occur.

(889) this * compiler has expired (Driver)
The demo period for this compiler has concluded.

(890) contact Microchip to purchase and re-activate this compiler (Driver)
The evaluation period of this demo installation of the compiler has expired. You will need to purchase the compiler to re-activate it. If you sincerely believe the evaluation period has ended prematurely, contact Microchip technical support.

(891) can't open psect usage map file "*": * (Driver)
The driver was unable to open the indicated file. The psect usage map file is generated by the driver when the driver option `--summary=`*`file`* is used. Ensure that the file is not open in another application.

(892) can't open memory usage map file "*": * (Driver)
The driver was unable to open the indicated file. The memory usage map file is generated by the driver when the driver option `--summary=`*`file`* is used. Ensure that the file is not open in another application.

(893) can't open HEX usage map file "*": * (Driver)
The driver was unable to open the indicated file. The HEX usage map file is generated by the driver when the driver option `--summary=`*`file`* is used. Ensure that the file is not open in another application.

(894) unknown source file type "*" (Driver)
The extension of the indicated input file could not be determined. Only files with the extensions `.as`, `.c`, `.obj`, `.usb`, `.p1`, `.lib` or `.hex` are identified by the driver.

(895) can't request and specify options in the one command (Driver)
The usage of the driver options `--getoption` and `--setoption` is mutually exclusive.

(896) no memory ranges specified for data space (Driver)
No on-chip or external memory ranges have been specified for the data space memory for the device specified.

(897) no memory ranges specified for program space (Driver)
No on-chip or external memory ranges have been specified for the program space memory for the device specified.

(899) can't open option file "*" for application "*": * (Driver)
An option file specified by a `--getoption` or `--setoption` driver option could not be opened. If you are using the `--setoption` option, ensure that the name of the file is spelled correctly and that it exists. If you are using the `--getoption` option ensure that this file can be created at the given location or that it is not in use by any other application.

(900) exec failed: * (Driver)
The subcomponent listed failed to execute. Does the file exist? Try re-installing the compiler.

(902) no chip name specified; use "* –CHIPINFO" to see available chip names (Driver)
The driver was invoked without selecting what chip to build for. Running the driver with the `-CHIPINFO` option will display a list of all chips that could be selected to build for.

(904) illegal format specified in "*" option (Driver)
The usage of this option was incorrect. Confirm correct usage with `--help` or refer to the part of the manual that discusses this option.

(905) illegal application specified in "*" option (Driver)
The application given to this option is not understood or does not belong to the compiler.

(907) unknown memory space tag "*" in "*" option specification (Driver)
A parameter to this memory option was a string but did not match any valid tags. Refer to the section of this manual that describes this option to see what tags (if any) are valid for this device.

(908) exit status = * (Driver)
One of the subcomponents being executed encountered a problem and returned an error code. Other messages should have been reported by the subcomponent to explain the problem that was encountered.

(913) "*" option can cause compiler errors in some standard header files (Driver)
Using this option will invalidate some of the qualifiers used in the standard header files, resulting in errors. This issue and its solution are detailed in the section of this manual that specifically discusses this option.

(915) no room for arguments (Preprocessor, Parser, Code Generator, Linker, Objtohex)
The code generator could not allocate any more memory.

(917) argument too long (Preprocessor, Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(918) *: no match (Preprocessor, Parser)
This is an internal compiler error. Contact Microchip Technical Support with details.

(919) * in chipinfo file "*" at line * (Driver)
The specified parameter in the chip configuration file is illegal.

(920) empty chipinfo file (Driver, Assembler)
The chip configuration file was able to be opened but it was empty. Try re-installing the compiler.

(922) chip “*” not present in chipinfo file “*” (Driver)
The chip selected does not appear in the compiler’s chip configuration file. Contact Microchip to see whether support for this device is available or it is necessary to upgrade the version of your compiler.

(923) unknown suboption “*” (Driver)
This option can take suboptions, but this suboption is not understood. This can just be a simple spelling error. If not,`--help` to look up what suboptions are permitted here.

(924) missing argument to “*” option (Driver)
This option expects more data but none was given. Check the usage of this option.

(925) extraneous argument to “*” option (Driver)
This option does not accept additional data, yet additional data was given. Check the usage of this option.

(926) duplicate “*” option (Driver)
This option can only appear once, but appeared more than once.

(928) bad “*” option value (Driver, Assembler)
The indicated option was expecting a valid hexadecimal integer argument.

(929) bad “*” option ranges (Driver)
This option was expecting a parameter in a range format (*`start_of_range-end_of_range`*), but the parameter did not conform to this syntax.

(930) bad “*” option specification (Driver)
The parameters to this option were not specified correctly. Run the driver with `--help` or refer to the driver’s chapter in this manual to verify the correct usage of this option.

(931) command file not specified (Driver)
Command file to this application, expected to be found after ’`@`’ or ’`<`’ on the command line was not found.

(939) no file arguments (Driver)
The driver has been invoked with no input files listed on its command line. If you are getting this message while building through a third party IDE, perhaps the IDE could not verify the source files to compile or object files to link and withheld them from the command line.

(940) *-bit * placed at * (Objtohex)
Presenting the result of the requested calculation.

(941) bad “*” assignment; USAGE: ** (Hexmate)
An option to Hexmate was incorrectly used or incomplete. Follow the usage supplied by the message and ensure that the option has been formed correctly and completely.

(942) unexpected character on line * of file “*” (Hexmate)
File contains a character that was not valid for this type of file, the file can be corrupt. For example, an Intel HEX file is expected to contain only ASCII representations of hexadecimal digits, colons (`:`) and line formatting. The presence of any other characters will result in this error.

(944) data conflict at address *h between * and * (Hexmate)
Sources to Hexmate request differing data to be stored to the same address. To force one data source to override the other, use the ‘`+`’ specifier. If the two named sources of conflict are the same source, then the source can contain an error.

(945) range (*h to *h) contained an indeterminate value (Hexmate)
The range for this calculation contained a value that could not be resolved. This can happen if the result was to be stored within the address range of the calculation.

(948) result width must be between 1 and 4 bytes (Hexmate)
The requested byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the `-CKSUM` option.

(949) start of range must be less than end of range (Hexmate)
The `-CKSUM` option has been given a range where the start is greater than the end. The parameters can be incomplete or entered in the wrong order.

(951) start of fill range must be less than end of range (Hexmate)
The `-FILL` option has been given a range where the start is greater than the end. The parameters can be incomplete or entered in the wrong order.

(953) unknown -HELP sub-option: * (Hexmate)
Invalid sub-option passed to `-HELP`. Check the spelling of the sub-option or use `-HELP` with no sub-option to list all options.

(956) -SERIAL value must be between 1 and * bytes long (Hexmate)
The serial number being stored was out of range. Ensure that the serial number can be stored in the number of bytes permissible by this option.

(958) too many input files specified; * file maximum (Hexmate)
Too many file arguments have been used. Try merging these files in several stages rather than in one command.

(960) unexpected record type (*) on line * of "*" (Hexmate)
Intel HEX file contained an invalid record type. Consult the Intel HEX format specification for valid record types.

(962) forced data conflict at address *h between * and * (Hexmate)
Sources to Hexmate force differing data to be stored to the same address. More than one source using the '`+`' specifier store data at the same address. The actual data stored there cannot be what you expect.

(963) range includes voids or unspecified memory locations (Hexmate)
The hash (checksum) range had gaps in data content. The runtime hash calculated is likely to differ from the compile-time hash due to gaps/unused byes within the address range that the hash is calculated over. Filling unused locations with a known value will correct this.

(964) unpaired nibble in -FILL value will be truncated (Hexmate)
The hexadecimal code given to the `-FILL` option contained an incomplete byte. The incomplete byte (nibble) will be disregarded.

(965) -STRPACK option not yet implemented; option will be ignored (Hexmate)
This option currently is not available and will be ignored.

(966) no END record for HEX file "*" (Hexmate)
Intel HEX file did not contain a record of type `END`. The HEX file can be incomplete.

(967) unused function definition "*" (from line *) (Parser)
The indicated `static` function was never called in the module being compiled. Being `static`, the function cannot be called from other modules so this warning implies the function is never used. Either the function is redundant, or the code that was meant to call it was excluded from compilation or misspelled the name of the function.

(968) unterminated string (Assembler)
A string constant appears not to have a closing quote.

(969) end of string in format specifier (Parser)
The format specifier for the `printf()` style function is malformed.

(970) character not valid at this point in format specifier (Parser)
The `printf()` style format specifier has an illegal character.

(971) type modifiers not valid with this format (Parser)
Type modifiers cannot be used with this format.

(972) only modifiers “h” and “l” valid with this format (Parser)
Only modifiers `h` (`short`) and `l` (`long`) are legal with this printf format specifier.

(973) only modifier “l” valid with this format (Parser)
The only modifier that is legal with this format is `l` (for `long`).

(974) type modifier already specified (Parser)
This type modifier has already be specified in this type.

(975) invalid format specifier or type modifier (Parser)
The format specifier or modifier in the printf-style string is illegal for this particular format.

(976) field width not valid at this point (Parser)
A field width cannot appear at this point in a `printf()` type format specifier.

(978) this identifier is already an enum tag (Parser)
This identifier following a `struct` or `union` keyword is already the tag for an enumerated type, and thus should only follow the keyword `enum`, for example:

```
enum IN {ONE=1, TWO};
struct IN {             /* oops -- IN is already defined */
    int a, b;
};
```

(979) this identifier is already a struct tag (Parser)
This identifier following a `union` or `enum` keyword is already the tag for a structure and should only follow the keyword `struct`, for example:

```
struct IN {
    int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(980) this identifier is already a union tag (Parser)
This identifier following a `struct` or `enum` keyword is already the tag for a `union` and should only follow the keyword `union`, for example:

```
union IN {
    int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(981) pointer required (Parser)
A pointer is required here, for example:

```
struct DATA data;
data->a = 9;      /* data is a structure, not a pointer to a structure */
```

(982) unknown op “*” in nxtuse() (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(983) storage class redeclared (Parser)
A variable previously declared as being `static` , has now be redeclared as `extern`.

(984) type redeclared (Parser)
The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration, for example:

```
int a;
char a;  /* oops -- what is the correct type? */
```

(985) qualifiers redeclared (Parser)
This function or variable has different qualifiers in different declarations.

(986) enum member redeclared (Parser)
A member of an enumeration is defined twice or more with differing values. Does the member appear twice in the same list or does the name of the member appear in more than one enum list?

(987) arguments redeclared (Parser)
The data types of the parameters passed to this function do not match its prototype.

(988) number of arguments redeclared (Parser)
The number of arguments in this function declaration does not agree with a previous declaration of the same function.

(989) module has code below file base of *h (Linker)
This module has code below the address given, but the `-C` option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing `psect` directives in assembler files.

(990) modulus by zero in #if; zero result assumed (Preprocessor)
A modulus operation in a `#if` expression has a zero divisor. The result has been assumed to be zero, for example:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
#define INTERESTING
#endif
```

(991) integer expression required (Parser)
In an `enum` declaration, values can be assigned to the members, but the expression must evaluate to a constant of type `int`, for example:

```
enum {one = 1, two, about_three = 3.12};
/* no non-int values allowed */
```

(992) can’t find op (Assembler)
This is an internal compiler error. Contact Microchip Technical Support with details.

(993) some command-line options are disabled (Driver)
The compiler is operating in demo mode. Some command-line options are disabled.

(994) some command-line options are disabled and compilation is delayed (Driver)
The compiler is operating in demo mode. Some command-line options are disabled, the compilation speed will be slower.

(995) some command-line options are disabled; code size is limited to 16kB, compilation is delayed (Driver)
The compiler is operating in demo mode. Some command-line options are disabled; the compilation speed will be slower, and the maximum allowed code size is limited to 16 KB.

8.3 Messages 1000 Thru 1499

(1015) missing “*” specification in chipinfo file “*” at line * (Driver)
This attribute was expected to appear at least once but was not defined for this chip.

(1016) missing argument* to “*” specification in chipinfo file “*” at line * (Driver)
This value of this attribute is blank in the chip configuration file.

(1017) extraneous argument* to “*” specification in chipinfo file “*” at line * (Driver)
There are too many attributes for the listed specification in the chip configuration file.

(1018) illegal number of “*” specification* (* found; * expected) in chipinfo file “*” at line * (Driver)
This attribute was expected to appear a certain number of times; but it did not appear for this chip.

(1019) duplicate “*” specification in chipinfo file “*” at line * (Driver)
This attribute can only be defined once, but has been defined more than once for this chip.

(1020) unknown attribute “*” in chipinfo file “*” at line * (Driver)
The chip configuration file contains an attribute that is not understood by this version of the compiler. Has the chip configuration file or the driver been replaced with an equivalent component from another version of this compiler?

(1021) syntax error reading “*” value in chipinfo file “*” at line * (Driver)
The chip configuration file incorrectly defines the specified value for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1022) syntax error reading “*” range in chipinfo file “*” at line * (Driver)
The chip configuration file incorrectly defines the specified range for this device. If you are modifying this file yourself, take care and refer to the comments at the beginning of this file for a description on what type of values are expected here.

(1024) syntax error in chipinfo file “*” at line * (Driver)
The chip configuration file contains a syntax error at the line specified.

(1025) unknown architecture in chipinfo file “*” at line * (Driver)
The attribute at the line indicated defines an architecture that is unknown to this compiler.

(1026) missing architecture in chipinfo file “*” at line * (Assembler)
The chipinfo file has a device section without an `ARCH` values. The architecture of the device must be specified. Contact Microchip Technical Support if the chipinfo file has not been modified.

(1027) activation was successful (Driver)
The compiler was successfully activated.

(1028) activation was not successful - error code (*) (Driver)
The compiler did not activated successfully.

(1029) compiler not installed correctly - error code (*) (Driver)
This compiler has failed to find any activation information and cannot proceed to execute. The compiler can have been installed incorrectly or incompletely. The error code quoted can help diagnose the reason for this failure. You can be asked for this failure code if contacting Microchip for assistance with this problem.

(1030) Hexmate - Intel HEX editing utility (Build 1.%i) (Hexmate)
Indicating the version number of the Hexmate being executed.

(1031) USAGE: * [input1.HEX] [input2.HEX]... [inputN.HEX] [options] (Hexmate)
The suggested usage of Hexmate.

(1032) use –HELP=<option> for usage of these command line options (Hexmate)
More detailed information is available for a specific option by passing that option to the `-HELP` option.

(1033) available command-line options: (Hexmate)
This is a simple heading that appears before the list of available options for this application.

(1034) type “*” for available options (Hexmate)
It looks like you need help. This advisory suggests how to get more information about the options available to this application or the usage of these options.

(1035) bad argument count (*) (Parser)
The number of arguments to a function is unreasonable. This is an internal compiler error. Contact Microchip Technical Support with details.

(1036) bad “*” optional header length (0x* expected) (Cromwell)
The length of the optional header in this COFF file was of an incorrect length.

(1037) short read on * (Cromwell)
When reading the type of data indicated in this message, it terminated before reaching its specified length.

(1038) string table length too short (Cromwell)
The specified length of the COFF string table is less than the minimum.

(1039) inconsistent symbol count (Cromwell)
The number of symbols in the symbol table has exceeded the number indicated in the COFF header.

(1040) bad : record 0x*, 0x* (Cromwell)
A record of the type specified failed to match its own value.

(1041) short record (Cromwell)
While reading a file, one of the file’s records ended short of its specified length.

(1042) unknown * record type 0x* (Cromwell)
The type indicator of this record did not match any valid types for this file format.

(1043) unknown optional header (Cromwell)
When reading this Microchip COFF file, the optional header within the file header was of an incorrect length.

(1044) end of file encountered (Cromwell, Linker)
The end of the file was found while more data was expected. Has this input file been truncated?

(1045) short read on block of * bytes (Cromwell)
A while reading a block of byte data from a UBROF record, the block ended before the expected length.

(1046) short string read (Cromwell)
A while reading a string from a UBROF record, the string ended before the specified length.

(1047) bad type byte for UBROF file (Cromwell)
This UBROF file did not begin with the correct record.

(1048) bad time/date stamp (Cromwell)
This UBROF file has a bad time/date stamp.

(1049) wrong CRC on 0x* bytes; should be * (Cromwell)
An end record has a mismatching CRC value in this UBROF file.

(1050) bad date in 0x52 record (Cromwell)
A debug record has a bad date component in this UBROF file.

(1051) bad date in 0x01 record (Cromwell)
A start of program record or segment record has a bad date component in this UBROF file.

(1052) unknown record type (Cromwell)
A record type could not be determined when reading this UBROF file.

(1053) additional RAM ranges larger than bank size (Driver)
A block of additional RAM being requested exceeds the size of a bank. Try breaking the block into multiple ranges that do not cross bank boundaries.

(1054) additional RAM range out of bounds (Driver)
The RAM memory range as defined through custom RAM configuration is out of range.

(1055) RAM range out of bounds (*) (Driver)
The RAM memory range as defined in the chip configuration file or through custom configuration is out of range.

(1056) unknown chip architecture (Driver)
The compiler is attempting to compile for a device of an architecture that is either unsupported or disabled.

(1058) assertion (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1059) rewrite loop (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1081) static initialization of persistent variable “*” (Parser, Code Generator)
A `persistent` variable has been assigned an initial value. This is somewhat contradictory as the initial value will be assigned to the variable during execution of the compiler’s startup code; however, the persistent qualifier requests that this variable shall be unchanged by the compiler’s startup code.

(1082) size of initialized array element is zero (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1088) function pointer “*” is used but never assigned a value (Code Generator)
A function call involving a function pointer was made, but the pointer was never assigned a target address, for example:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1089) recursive function call to “*” (Code Generator)
A recursive call to the specified function has been found. The call can be direct or indirect (using function pointers) and can be either a function calling itself, or calling another function whose call graph includes the function under consideration.

(1090) variable “*” is not used (Code Generator)
This variable is declared but has not been used by the program. Consider removing it from the program.

(1091) main function “*” not defined (Code Generator)
The main function has not been defined. Every C program must have a function called `main()`.

(1094) bad derived type (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1095) bad call to typeSub() (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1096) type should be unqualified (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1097) unknown type string “*” (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1098) conflicting declarations for variable “*” (*:*) (Parser, Code Generator)
Differing type information has been detected in the declarations for a variable, or between a declaration and the definition of a variable, for example:

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1118) bad string “*” in getexpr(J) (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1119) bad string “*” in getexpr(LRN) (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1121) expression error (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1137) match() error: * (Code Generator)
This is an internal compiler error. Contact Microchip Technical Support with details.

(1157) W register must be W9 (Assembler)
The working register required here has to be W9, but an other working register was selected.

(1159) W register must be W11 (Assembler)
The working register required here has to be W11, but an other working register was selected.

(1178) the "*" option has been removed and has no effect (Driver)
This option no longer exists in this version of the compiler and has been ignored. Use the compiler's `--help` option or refer to the manual to find a replacement option.

(1179) interrupt level for function "*" cannot exceed * (Code Generator)
The interrupt level for the function specified is too high. Each interrupt function is assigned a unique interrupt level. This level is considered when analyzing the call graph and reentrantly called functions. If using the `interrupt_level` pragma, check the value specified.

(1180) directory "*" does not exist (Driver)
The directory specified in the setup option does not exist. Create the directory and try again.

(1182) near variables must be global or static (Code Generator)
A variable qualified as `near` must also be qualified with `static` or made global. An auto variable cannot be qualified as near.

(1183) invalid version number (Activation)
During activation, no matching version number was found on the Microchip activation server database for the serial number specified.

(1184) activation limit reached (Activation)
The number of activations of the serial number specified has exceeded the maximum number allowed for the license.

(1185) invalid serial number (Activation)
During activation, no matching serial number was found on the Microchip activation server database.

(1186) license has expired (Driver)
The time-limited license for this compiler has expired.

(1187) invalid activation request (Driver)
The compiler has not been correctly activated.

(1188) network error * (Activation)
The compiler activation software was unable to connect to the Microchip activation server via the network.

(1190) FAE license only - not for use in commercial applications (Driver)
Indicates that this compiler has been activated with an FAE license. This license does not permit the product to be used for the development of commercial applications.

(1191) licensed for educational use only (Driver)
Indicates that this compiler has been activated with an education license. The educational license is only available to educational facilities and does not permit the product to be used for the development of commercial applications.

(1192) licensed for evaluation purposes only (Driver)
Indicates that this compiler has been activated with an evaluation license.

(1193) this license will expire on * (Driver)
The compiler has been installed as a time-limited trial. This trial will end on the date specified.

(1195) invalid syntax for "*" option (Driver)
A command line option that accepts additional parameters was given inappropriate data or insufficient data. For example, an option can expect two parameters with both being integers. Passing a string as one of these parameters or supplying only one parameter could result in this error.

(1198) too many "*" specifications; * maximum (Hexmate)
This option has been specified too many times. If possible, try performing these operations over several command lines.

(1199) compiler has not been activated (Driver)
The trial period for this compiler has expired. The compiler is now inoperable until activated with a valid serial number. Contact Microchip to purchase this software and obtain a serial number.

(1200) Found %0*lXh at address *h (Hexmate)
The code sequence specified in a `-FIND` option has been found at this address.

(1201) all FIND/REPLACE code specifications must be of equal width (Hexmate)
All find, replace and mask attributes in this option must be of the same byte width. Check the parameters supplied to this option. For example, finding 1234h (2 bytes) masked with FFh (1 byte) results in an error; but, masking with 00FFh (2 bytes) works.

(1202) unknown format requested in -FORMAT: * (Hexmate)
An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated (Hexmate)
Data to this option was not entered as whole bytes. Perhaps the data was incomplete or a leading zero was omitted. For example, the value Fh contains only four bits of significant data and is not a whole byte. The value 0Fh contains eight bits of significant data and is a whole byte.

(1204) * value must be between 1 and * bytes long (Hexmate)
An illegal length of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

(1205) using the configuration file *; you can override this with the environment variable HTC_XML (Driver)
This is the compiler configuration file selected during compiler setup. This can be changed via the `HTC_XML` environment variable. This file is used to determine where the compiler has been installed.

(1207) some of the command line options you are using are now obsolete (Driver)
Some of the command line options passed to the driver have now been discontinued in this version of the compiler; however, during a grace period these old options will still be processed by the driver.

(1208) use –help option or refer to the user manual for option details (Driver)
An obsolete option was detected. Use `--help` or refer to the manual to find a replacement option that will not result in this advisory message.

(1209) An old MPLAB tool suite plug-in was detected. (Driver)
The options passed to the driver resemble those that the Microchip MPLAB 8 IDE would pass to a previous version of this compiler. Some of these options are now obsolete – however, they were still interpreted. It is recommended that you install an updated Microchip options plug-in for the IDE.

(1210) Visit the Microchip website (www.microchip.com) for a possible upgrade (Driver)
Visit our website to see if an upgrade is available to address the issue(s) listed in the previous compiler message. Navigate to the MPLAB XC8 C Compiler page and look for a version upgrade downloadable file. If your version is current, contact Microchip Technical Support for further information.

(1212) Found * (%0*lXh) at address *h (Hexmate)
The code sequence specified in a `-FIND` option has been found at this address.

(1213) duplicate ARCH for * in chipinfo file at line * (Assembler, Driver)
The chipinfo file has a device section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact Microchip Technical Support with details.

(1218) can't create cross reference file * (Assembler)
The assembler attempted to create a cross reference file; but it could not be created. Check that the file's path name is correct.

(1228) unable to locate installation directory (Driver)
The compiler cannot determine the directory where it has been installed.

(1230) dereferencing uninitialized pointer "*" (Code Generator)
A pointer that has not yet been assigned a value has been dereferenced. This can result in erroneous behavior at runtime.

(1235) unknown keyword * (Driver)
The token contained in the USB descriptor file was not recognized.

(1236) invalid argument to *: * (Driver)
An option that can take additional parameters was given an invalid parameter value. Check the usage of the option or the syntax or range of the expected parameter.

(1237) endpoint 0 is pre-defined (Driver)
An attempt has been made to define endpoint 0 in a USB file.

(1238) FNALIGN failure on * (Linker)
Two functions have their auto/parameter blocks aligned using the `FNALIGN` directive, but one function calls the other, which implies that must not be aligned. This will occur if a function pointer is assigned the address of each function, but one function calls the other. For example:

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* ! */
int (*ip)(int);
ip = one;
ip(23);
ip = two;      /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets (Code Generator)
A function call involving a function pointer was made, but the pointer was never assigned a target address, for example:

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1240) unknown algorithm type (%i) (Driver)
The error file specified after the `-Efile` or `-E+file` options could not be opened. Check to ensure that the file or directory is valid and that has read only access.

(1241) bad start address in * (Driver)
The start of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1242) bad end address in * (Driver)
The end of range address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1243) bad destination address in * (Driver)
The destination address for the `--CHECKSUM` option could not be read. This value must be a hexadecimal number.

(1245) value greater than zero required for * (Hexmate)
The align operand to the Hexmate `-FIND` option must be positive.

(1246) no RAM defined for variable placement (Code Generator)
No memory has been specified to cover the banked RAM memory.

(1247) no access RAM defined for variable placement (Code Generator)
No memory has been specified to cover the access bank memory.

(1248) symbol (*) encountered with undefined type size (Code Generator)
The code generator was asked to position a variable, but the size of the variable is not known. This is an internal compiler error. Contact Microchip Technical Support with details.

(1250) could not find space (* byte*) for variable * (Code Generator)
The code generator could not find space in the banked RAM for the variable specified.

(1253) could not find space (* byte*) for auto/param block (Code Generator)
The code generator could not find space in RAM for the psect that holds auto and parameter variables.

(1254) could not find space (* byte*) for data block (Code Generator)
The code generator could not find space in RAM for the data psect that holds initialized variables.

(1255) conflicting paths for output directory (Driver)
The compiler has been given contradictory paths for the output directory via any of the `-O` or `--OUTDIR` options, for example:

```
--outdir=../../   -o../main.HEX
```

(1256) undefined symbol “*” treated as HEX constant (Assembler)
A token which could either be interpreted as a symbol or a hexadecimal value does not match any previously defined symbol and so will be interpreted as the latter. Use a leading zero to avoid the ambiguity, or use an alternate radix specifier such as `0x`. For example:

```
MOV  a, F7h  ; is this the symbol F7h, or the HEX number 0xF7?
```

(1257) local variable “*” is used but never given a value (Code Generator)
An auto variable has been defined and used in an expression, but it has not been assigned a value in the C code before its first use. Auto variables are not cleared on startup and their initial value is undefined. For example:

```
int main(void) {
    double src, out;
    out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function “*” (Code Generator)
The call tree analysis by the code generator indicates that the hardware stack can overflow. This should be treated as a guide only. Interrupts, the assembler optimizer and the program structure can affect the stack usage. The stack usage is based on the C program and does not include any call tree derived from assembly code.

(1259) can’t optimize for both speed and space (Driver)
The driver has been given contradictory options of compile for speed and compile for space, for example:

```
--opt=speed,space
```

(1260) macro “*” redefined (Assembler)
More than one definition for a macro with the same name has been encountered, for example:

```
MACRO fin
    ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
    reti
ENDM
```

(1261) string constant required (Assembler)
A string argument is required with the `DS` or `DSU` directive, for example:

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1262) object “*” lies outside available * space (Code Generator)
An absolute variable was positioned at a memory location which is not within the memory defined for the target device, for example:

```
int data __at(0x800);     /* oops -- is this the correct address? */
```

(1264) unsafe pointer conversion (Code Generator)
A pointer to one kind of structure has been converted to another kind of structure and the structures do not have a similar definition, for example:

```
struct ONE {
    unsigned a;
    long b;          /* ! */
} one;
struct TWO {
    unsigned a;
    unsigned b;      /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;    /* oops -- was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * into * bytes at 0x* (Linker)
See error message 1356 for more information.

(1268) fixup overflow storing 0x* in * bytes at * (Linker)
See error message 1356 for more information.

(1273) Omniscient Code Generation not available in Free mode (Driver)
This message advises that advanced features of the compiler are not be enabled in this Free mode compiler.

(1275) the qualifier “*” is only applicable to functions (Parser)
A qualifier which only makes sense when used in a function definition has been used with a variable definition.

```
interrupt int dacResult;  /* oops --
the interrupt qualifier can only be used with functions */
```

(1276) buffer overflow in DWARF location list (Cromwell)
A buffer associated with the ELF/DWARF debug file has overflowed. Contact Microchip Technical Support with details.

(1278) omitting “*” which does not have a location (Cromwell)
A variable has no storage location listed and will be omitted from the debug output. Contact Microchip Technical Support with details.

(1284) malformed mapfile while generating summary: CLASS expected but not found (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1285) malformed mapfile while generating summary: no name at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1286) malformed mapfile while generating summary: no link address at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1287) malformed mapfile while generating summary: no load address at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1288) malformed mapfile while generating summary: no length at position * (Driver)
The map file being read to produce a memory summary is malformed. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1289) line range limit exceeded, possibly affecting ability to debug code (Cromwell)
A C statement has produced assembly code output whose length exceeds a preset limit. This means that debug information produced by CROMWELL may not be accurate. This warning does not indicate any potential code failure.

(1290) buffer overflow in DWARF debugging information entry (Cromwell)
A buffer associated with the ELF/DWARF debug file has overflowed. Contact Microchip Technical Support with details.

(1291) bad ELF string table index (Cromwell)
An ELF file passed to CROMWELL is malformed and cannot be used.

(1292) malformed define in .SDB file * (Cromwell)
The named SDB file passed to CROMWELL is malformed and cannot be used.

(1293) couldn't find type for "*" in DWARF debugging information entry (Cromwell)
The type of symbol could not be determined from the SDB file passed to CROMWELL. Either the file has been edited or corrupted, or this is a compiler error – contact Microchip Technical Support with details.

(1294) there is only one day left until this license expires (Driver)
The compiler is running as a demo and will be unable to run in PRO mode after the evaluation license has expired in less than one day's time. After expiration, the compiler can be operated in Free mode indefinitely, but will produce a larger output binary.

(1295) there are * days left until this license will expire (Driver)
The compiler is running as a demo and will be unable to run in PRO mode after the evaluation license has expired in the indicated time. After expiration, the compiler can be operated in Free mode indefinitely, but will produce a larger output binary.

(1296) source file "*" conflicts with "*" (Driver)
The compiler has encountered more than one source file with the same base name. This can only be the case if the files are contained in different directories. As the compiler and IDEs based the names of intermediate files on the base names of source files, and intermediate files are always stored in the same location, this situation is illegal. Ensure the base name of all source files are unique.

(1297) option * not available in Free mode (Driver)
Some options are not available when the compiler operates in Free mode. The options disabled are typically related to how the compiler is executed, e.g., `--getoption` and `--setoption`, and do not control compiler features related to code generation.

(1298) use of * outside macros is illegal (Assembler)
Some assembler directives, e.g., `EXITM`, can only be used inside macro definitions.

(1299) non-standard modifier "*" - use "*" instead (Parser)
A printf placeholder modifier has been used which is non-standard. Use the indicated modifier instead. For example, the standard `hh` modifier should be used in preference to `b` to indicate that the value should be printed as a `char` type.

(1300) maximum number of program classes reached; some classes may be excluded from debugging information (Cromwell)
CROMWELL is passed a list of class names on the command line. If the number of class names passed in is too large, not all will be used and there is the possibility that debugging information will be inaccurate.

(1301) invalid ELF section header; skipping (Cromwell)
CROMWELL found an invalid section in an ELF section header. This section will be skipped.

(1302) could not find valid ELF output extension for this device (Cromwell)
The extension could not be for the target device family.

(1303) invalid variable location detected: * - * (Cromwell)
A symbol location could not be determined from the SDB file.

(1304) unknown register name: "*" (Cromwell)
The location for the indicated symbol in the SDB file was a register, but the register name was not recognized.

(1305) inconsistent storage class for variable: “*” (Cromwell)
The storage class for the indicated symbol in the SDB file was not recognized.

(1306) inconsistent size (* vs *) for variable: “*” (Cromwell)
The size of the symbol indicated in the SDB file does not match the size of its type.

(1307) psect * truncated to * bytes (Driver)
The psect representing either the stack or heap could not be made as large as requested and will be truncated to fit the available memory space.

(1308) missing/conflicting interrupts sub-option; defaulting to “*” (Driver)
The suboptions to the `--INTERRUPT` option are missing or malformed, for example:

```
--INTERRUPTS=single,multi
```

Oops, did you mean single-vector or multi-vector interrupts?

(1309) ignoring invalid runtime * sub-option (*) using default (Driver)
The indicated suboption to the `--RUNTIME` option is malformed, for example:

```
--RUNTIME=default,speed:0y1234
```

Oops, that should be 0x1234.

(1310) specified speed (*Hz) exceeds max operating frequency (*Hz); defaulting to *Hz (Driver)
The frequency specified to the perform suboption to `--RUNTIME` option is too large for the selected device.

```
--RUNTIME=default,speed:0xffffffff
```

Oops, that value is too large.

(1311) missing configuration setting for config word *; using default (Driver)
The configuration settings for the indicated word have not be supplied in the source code and a default value will be used.

(1312) conflicting runtime perform sub-option and configuration word settings; assuming *Hz (Driver)
The configuration settings and the value specified with the perform suboption of the `--RUNTIME` options conflict and a default frequency has been selected.

(1313) * sub-options (“*”) ignored (Driver)
The argument to a suboption is not required and will be ignored.

```
--OUTPUT=intel:8
```

Oops, the :8 is not required

(1314) illegal action in memory allocation (Code Generator)
This is an internal error. Contact Microchip Technical Support with details.

(1315) undefined or empty class used to link psect * (Linker)
The linker was asked to place a psect within the range of addresses specified by a class, but the class was either never defined, or contains no memory ranges.

(1316) attribute “*” ignored (Parser)
An attribute has been encountered that is valid, but which is not implemented by the parser. It will be ignored by the parser and the attribute will have no effect. Contact Microchip Technical Support with details.

(1317) missing argument to attribute “*” (Parser)
An attribute has been encountered that requires an argument, but this is not present. Contact Microchip Technical Support with details.

(1318) invalid argument to attribute “*” (Parser)
An argument to an attribute has been encountered, but it is malformed. Contact Microchip Technical Support with details.

(1319) invalid type “*” for attribute “*” (Parser)
This indicated a bad option passed to the parser. Contact Microchip Technical Support with details.

(1320) attribute “*” already exists (Parser)
This indicated the same attribute option being passed to the parser more than once. Contact Microchip Technical Support with details.

(1321) bad attribute -T option “%s” (Parser)
The attribute option passed to the parser is malformed. Contact Microchip Technical Support with details.

(1322) unknown qualifier “%s” given to -T (Parser)
The qualifier specified in an attribute option is not known. Contact Microchip Technical Support with details.

(1323) attribute expected (Parser)
The `__attribute__` directive was used but did not specify an attribute type.

```
int rv (int a) __attribute__(())  /* oops -- what is the attribute? */
```

(1324) qualifier “*” ignored (Parser)
Some qualifiers are valid, but cannot be implemented on some compilers or target devices. This warning indicates that the qualifier will be ignored.

(1342) whitespace after “\” (Preprocessor)
Whitespace characters have been found between a backslash and newline characters and will be ignored.

(1343) hexfile data at address 0x* (0x*) overwritten with 0x* (Objtohex)
The indicated address is about to be overwritten by additional data. This would indicate more than one section of code contributing to the same address.

(1346) can’t find 0x* words for psect “*” in segment “*” (largest unused contiguous range 0x%lX) (Linker)
See also message (491). The new form of message also indicates the largest free block that the linker could find. Unless there is a single space large enough to accommodate the psect, the linker will issue this message. Often when there is banking or paging involved the largest free space is much smaller than the total amount of space remaining,

(1347) can’t find 0x* words (0x* withtotal) for psect “*” in segment “*” (largest unused contiguous range 0x%lX) (Linker)
See also message (593). The new form of message also indicates the largest free block that the linker could find. Unless there is a single space large enough to accommodate the psect, the linker will issue this message. Often when there is banking or paging involved the largest free space is much smaller than the total amount of space remaining,

(1348) enum tag “*” redefined (from *:*) (Parser)
More than one `enum` tag with the same name has been defined, The previous definition is indicated in the message.

```
enum VALS { ONE=1, TWO, THREE };
enum VALS { NINE=9, TEN };  /* oops -- is VALS the right tag name? */
```

(1349) initialization of absolute variable "*" in RAM is not supported (Code Generator)
An absolute variable, one defined using `__at(address)`, cannot be assigned an initial value when it is defined. Place an assignment to the variable at an appropriate location in your program.

```
int foobar __at(0x20) = 0x55;  /* oops --
 you cannot assign a value to an absolute variable */
```

(1350) pointer operands to “-” must reference the same array (Code Generator)
If two addresses are subtracted, the addresses must be of the same object to be ANSI compliant.

```
int * ip;
int fred, buf[20];
ip = &buf[0] - &fred;  /* oops --
 second operand must be an address of a "buf" element */
```

(1352) truncation of operand value (0x*) to * bits (Assembler)
The operand to an assembler instruction was too large and was truncated.

```
movlw  0x321  ; oops -- is this the right value?
```

(1354) ignoring configuration setting for unimplemented word * (Driver)
A Configuration Word setting was specified for a Word that does not exist on the target device.

```
__CONFIG(3, 0x1234);  /* config word 3 does not exist on an 18C801 */
```

(1355) in-line delay argument too large (Code Generator)
The in-line delay sequence `_delay` has been used, but the number of instruction cycles requested is too large. Use this routine multiple times to achieve the desired delay length.

```
#include <xc.h>
int main(void) {
    delay(0x400000); /* oops -- cannot delay by this number of cycles */
}
```

(1356) fixup overflow referencing * * (0x*) into * byte* at 0x*/0x* -> 0x* (* */0x*) (Linker)**
’Fixup’ is the process conducted by the linker of replacing symbolic references to operands with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory. ‘Fixup overflow’ is when a symbol’s value is too large to fit within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol used to represent this address has the value 0x110, then clearly this value cannot be encoded into the instruction.

Fixup errors are often caused by hand-written assembly code. Common mistakes that trigger these errors include failing to mask a full, banked data address in file register instructions, or failing to mask the destination address in jump or call instructions. If this error is triggered by assembly code generated from C source, then it is often that constructs like `switch()` statements have generated a block of assembly too large for jump instructions to span. Adjusting the default linker options can also causes such errors.

To identify these errors, follow these steps.

- Perform a debug build (in MPLAB X IDE select **Debug** > **Discrete Debugger Operation** > **Build for Debugging**; alternatively, on the command line use the `-D__DEBUG` option).

- Open the relevant assembler list file (ensure the MPLAB X IDE project properties has **XC8 Compiler** > **Preprocessing and Messaging** > **Generate the ASM listing file** enabled; alternatively, on the command line, use the `-Wa,-a` option).
- Find the instruction at the address quoted in the error message.

Consider the following error message.

```
main.c: 4: (1356)(linker) fixup overflow referencing psect bssBANK1 (0x100) into 1
byte at 0x7FF0/0x1 -> 0x7FF0 (main.obj 23/0x0)
```

The file being linked was `main.obj`. This tells you the assembly list file in which you should be looking is `main.lst`. The location of the instruction at fault is 0x7FF0 (You can also tell from this message that the instruction is expecting a 1 byte quantity—this size is rounded to the nearest byte—but the value was determined to be 0x100).

In the assembly list file, search for the address specified in the error message.

```
61   007FF0   6F00      movwf   _foobar,b       ;#
```

and to confirm, look for the symbol referenced in the assembler instruction at this address in the symbol table at the bottom of the same file.

```
Symbol Table                    Tue Oct 28 11:06:37 2014
_foobar 0100
```

In this example, the hand-written PIC18 `movwf` instruction causing the problem takes an 8-bit offset into a bank of memory, but clearly the address 0x100 exceeds this size. The instruction should have been written as:

```
MOVWF        BANKMASK(_foo)
```

which masks out the top bits of the address containing the bank information. Alternatively, the linker can be instructed to ignore fixup overflows, or to issue a warning rather than an error in such situations. See 6.1.31. Fixupoverflow Linker Option for more information, as the use of this option may compromise the detection of genuine errors.

If the assembler instruction that caused this error was generated by the compiler, in the assembler list file look back up the file from the instruction at fault to determine which C statement has generated this instruction. You will then need to examine the C code for possible errors.

(1357) fixup overflow storing 0x* in * byte* at 0x*/0x* -> 0x* (* */0x*) (Linker)**
See message (1356).

(1358) no space for * temps (*) (Code Generator)
The code generator was unable to find a space large enough to hold the temporary variables (scratch variables) for this program.

(1359) no space for * parameters (Code Generator)
The code generator was unable to find a space large enough to hold the parameter variables for a particular function.

(1360) no space for auto/param * (Code Generator)
The code generator was unable to find a space large enough to hold the auto variables for a particular function. Some parameters passed in registers can need to be allocated space in this auto area as well.

(1361) syntax error in configuration argument (Parser)
The argument to `#pragma config` was malformed.

```
#pragma config WDT     /* oops -- is WDT on or off? */
```

(1362) configuration setting *=* redefined (Code Generator)
The same `config` pragma setting have been issued more than once with different values.

```
#pragma config WDT=OFF
#pragma config WDT=ON     /* oops -- is WDT on or off? */
```

(1363) unknown configuration setting (* = *) used (Driver)
The configuration value and setting is not known for the target device. The use of an unknown configuration register number may also trigger this message.

```
#pragma config WDR=ON     /* oops -- did you mean WDT? */
#pragma config CONFIG1L=0x46 /* oops -- no 1L register on a 18F4520 */
```

(1364) can’t open configuration registers data file * (Driver)
The file containing value configuration settings could not be found.

(1365) missing argument to pragma “varlocate” (Parser)
The argument to `#pragma varlocate` was malformed.

```
#pragma varlocate    /* oops -- what do you want to locate & where? */
```

(1366) syntax error in pragma “varlocate” (Parser)
The argument to `#pragma varlocate` was malformed.

```
#pragma varlocate fred     /* oops -- which bank for fred? */
```

(1367) end of file in _asm (Parser)
An end-of-file marker was encountered inside a `_asm _endasm` block.

(1368) assembler message: * (Assembler)
Displayed is an assembler advisory message produced by the `MESSG` directive contained in the assembler source.

(1369) can’t open proc file * (Driver)
The proc file for the selected device could not be opened.

(1370) peripheral library support is not available for the * (Driver)
The peripheral library is not available for the selected device.

(1371) float type can’t be bigger than double type; double has been changed to * bits (Driver)
Use of the `-fshort-double` options has result in the size of the `double` type being smaller than that of the `float` type. This is not permitted by the C Standard. The `double` type size has been increased to be that indicated.

(1372) interrupt level cannot be greater than * (Code Generator)
The specific `interrupt_level` is too high for the device selected.

```
#pragma interrupt_level 4
// oops - there aren't that many interrupts on this device
```

(1374) the compiler feature “*” is no longer supported; * (Driver)
The feature indicated is no longer supported by the compiler.

(1375) multiple interrupt functions (* and *) defined for device with only one interrupt vector (Code Generator)
The named functions have both been qualified interrupt, but the target device only supports one interrupt vector and hence one interrupt function.

```
interrupt void isr_lo(void) {
// ...
}
interrupt void isr_hi(void) {   // oops, cannot define two ISRs
// ...
}
```

(1376) initial value (*) too large for bitfield width (*) (Code Generator)
A structure with bit-fields has been defined an initialized with values. The value indicated it too large to fit in the corresponding bit-field width.

```
struct {
unsigned flag :1;
unsigned mode :3;
} foobar = { 1, 100 };   // oops, 100 is too large for a 3 bit object
```

(1377) no suitable strategy for this switch (Code Generator)
The compiler was unable to determine the switch strategy to use to encode a C `switch` statement based on the code and your selection using the `#pragma switch` directive. You can need to choose a different strategy.

(1378) syntax error in pragma “*” (Parser)
The arguments to the indicated pragma are not valid.

```
#pragma addrqual ingore  // oops -- did you mean ignore?
```

(1379) no suitable strategy for this switch (Code Generator)
The compiler encodes `switch()` statements according to one of a number of strategies. The specific number and values of the case values, and the switch expression, as well as the `switch` pragma determine the strategy chosen. This error indicates that no strategy was available to encode the `switch()` statement. Contact Microchip support with program details.

(1380) unable to use switch strategy “*” (Code Generator)
The compiler encodes `switch()` statements according to one of a number of strategies. The specific number and values of the case values, and the switch expression, as well as the `switch` pragma, determine the strategy chosen. This error indicates that the strategy that was requested cannot be used to encode the `switch()` statement. Contact Microchip support with program details.

(1381) invalid case label range (Parser)
The values supplied for the case range are not correct. They must form an ascending range and be integer constants.

```
case 0 ... -2:  // oops -- do you mean -2 ... 0  ?
```

(1385) * “*” is deprecated (declared at *:*) (Parser)
Code is using a variable or function that was marked as being deprecated using an attribute.

```
char __attribute__((deprecated)) foobar;
foobar = 9;     // oops -- this variable is near end-of-life
```

(1386) unable to determine the semantics of the configuration setting "*" for register "*" (Parser, Code Generator)

The numerical value supplied to a configuration bit setting has no direct association setting specified in the data sheet. The compiler will attempt to honor your request, but check your device data sheet.

```
#pragma config OSC=11
// oops -- there is no direct association for that value on an 18F2520
// either use OSC=3 or OSC=RC
```

(1387) in-line delay argument must be constant (Code Generator)

The `_delay` in-line function can only take a constant expression as its argument.

```
int delay_val = 99;
_delay(delay_val);   // oops, argument must be a constant expression
```

(1388) configuration setting/register of "*" with 0x* will be truncated by 0x* (Parser, Code Generator)

A Configuration bit has been programmed with a value that is either too large for the setting, or is not one of the prescribed values.

```
#pragma config WDTPS=138  // oops -- do you mean 128?
```

(1389) attempt to reprogram configuration * "*" with * (is *) (Parser, Code Generator)

A Configuration bit that was already programmed has been programmed again with a conflicting setting to the original.

```
#pragma config WDT=ON
#pragma config WDT=OFF  // oops -- watchdog on or off?
```

(1390) identifier specifies insignificant characters beyond maximum identifier length (Parser)

An identifier that has been used is so long that it exceeds the set identifier length. This can mean that long identifiers cannot be correctly identified and the code will fail. The maximum identifier length can be adjusted using the -N option.

```
int theValueOfThePortAfterTheModeBitsHaveBeenSet;
// oops, make your symbol shorter or increase the maximum
// identifier length
```

(1391) constant object size of * exceeds the maximum of * for this chip (Code Generator)

The const object defined is too large for the target device.

```
const int array[200] = { ... };  // oops -- not on a Baseline part!
```

(1392) function "*" is called indirectly from both mainline and interrupt code (Code Generator)

A function has been called by main-line (non-interrupt) and interrupt code. If this warning is issued, it highlights that such code currently violates a compiler limitation for the selected device.

(1393) possible hardware stack overflow detected; estimated stack depth: * (Code Generator)

The compiler has detected that the call graph for a program could be using more stack space that allocated on the target device. If this is the case, the code can fail. The compiler can only make assumption regarding the stack usage, when interrupts are involved and these lead to a worst-case estimate of stack usage. Confirm the function call nesting if this warning is issued.

(1394) attempting to create memory range (* - *) larger than page size * (Driver)
The compiler driver has detected that the memory settings include a program memory “page” that is larger than the page size for the device. This would mostly likely be the case if the --ROM option is used to change the default memory settings. Consult your device data sheet to determine the page size of the device you are using and to ensure that any contiguous memory range you specify using the --ROM option has a boundary that corresponds to the device page boundaries.

```
--ROM=100-1fff
```

The above might need to be paged. If the page size is 800h, the above could specified as

```
--ROM=100-7ff,800-fff,1000-17ff,1800-1fff
```

(1395) notable code sequence candidate suitable for compiler validation suite detected (*) (Code Generator)
The compiler has in-built checks that can determine if combinations of internal code templates have been encountered. Where unique combinations are uncovered when compiling code, this message is issued. This message is not an error or warning and its presence does not indicate possible code failure, but if you are willing to participate, the code you are compiling can be sent to Support to assist with the compiler testing process.

(1396) “*” positioned in the * memory region (0x* - 0x*) reserved by the compiler (Code Generator)
Some memory regions are reserved for use by the compiler. These regions are not normally used to allocate variables defined in your code. However, by making variables absolute, it is possible to place variables in these regions and avoid errors that would normally be issued by the linker. Absolute variables can be placed at any location, even on top of other objects. This warning from the code generator indicates that an absolute has been detected that will be located at memory that the compiler will be reserving. You must locate the absolute variable at a different location. This message will commonly be issued when placing variables in the common memory space.

```
char shared __at(0x7);   // oops, this memory is required by the compiler
```

(1397) unable to implement non-stack call to “*”; possible hardware stack overflow (Code Generator)
The compiler must encode a C function call without using a call assembly instruction and the hardware stack (i.e., use a lookup table), but is unable to. A call instruction might be required if the function is called indirectly via a pointer, but if the hardware stack is already full, an additional call will cause a stack overflow.

(1401) eeprom qualified variables can’t be accessed from both interrupt and mainline code (Code Generator)
All `eeprom` variables are accessed via routines that are not reentrant. Code might fail if an attempt is made to access `eeprom`-qualified variables from interrupt and main-line code. Avoid accessing `eeprom` variables in interrupt functions.

(1402) a pointer to eeprom can’t also point to other data types (Code Generator)
A pointer cannot have targets in both the EEPROM space and ordinary data space.

(1403) pragma “*” ignored (Parser)
The pragma you have specified has no effect and will be ignored by the compiler. This message can only be issued in C18 compatibility mode.

```
#pragma varlocate "mySection" fred  // oops -- not accepted
```

(1404) unsupported: * (Parser)
The unsupported `__attribute__` has been used to indicate that some code feature is not supported.

The message printed will indicate the feature that is not supported and which should be avoided.

(1405) storage class specifier “*” ignored (Parser)
The storage class you have specified is not required and will be ignored by the compiler. This message can only be issued in C18 compatibility mode.

```
int procInput(auto int inValue)  // oops -- no need for auto
{ ...
```

(1406) auto eeprom variables are not supported (Code Generator)
Variables qualified as `eeprom` cannot be auto. You can define `static` local objects qualified as `eeprom`, if required.

```
int main(void) {
eeprom int mode;  // oops -- make this static or global
```

(1407) bit eeprom variables are not supported (Code Generator)
Variables qualified as `eeprom` cannot have type `bit`.

```
eeprom bit myEEbit;    // oops -- you cannot define bits in EEPROM
```

(1408) ignoring initialization of far variables (Code Generator)
Variables qualified as `far` cannot be assigned an initial value. Assign the value later in the code.

```
far int chan = 0x1234; // oops -- you cannot assign a value here
```

(1409) warning number used with pragma “warning” is invalid (Parser)
The message number used with the `warning` pragma is below zero or larger than the highest message number available.

```
#pragma warning disable 1316 13350   // oops -- possibly number 1335?
```

(1410) can’t assign the result of an invalid function pointer (Code Generator)
The compiler allows some functions to be called via a constant cast to be a function pointer, but not all. The address specified is not valid for this device.

```
foobar += ((int (*)(int))0x0)(77);
// oops -- you cannot call a function with a NULL pointer
```

(1411) Additional ROM range out of bounds (Driver)
Program memory specified with the `-mrom` option is outside of the on-chip, or external, memory range supported by this device.

```
-mrom=default,+2000-2ffff
```

Oops -- memory too high, should that be 2fff?

(1412) missing argument to pragma “warning disable” (Parser)
Following the `#pragma warning` disable should be a comma-separated list of message numbers to disable.

```
#pragma warning disable  // oops -- what messages are to be disabled?
```

Try something like the following.

```
#pragma warning disable 1362
```

(1413) pointer comparisons involving address of "*", positioned at address 0x0, may be invalid (Code Generator)
An absolute object placed at address 0 has had its address taken. By definition, this is a `NULL` pointer and code which checks for `NULL` (i.e., checks to see if the address is valid) can fail.

```
int foobar __at(0x00);
int  * ip;
int main(void)
{
    ip = &foobar;  // oops -- 0 is not a valid address
```

(1414) option * is defunct and has no effect (Driver)
The option used is now longer supported. It will be ignored.

```
xc8 --chip=18f452 --cp=24 main.c
```

Oops -- the `--cp` option is no longer required.

(1415) argument to "merge" psect flag must be 0 or 1 (Assembler)
This psect flag must be assigned a 0 or 1.

```
PSECT myTxt,class=CODE,merge=true  ; oops -- I think you mean merge=1
```

(1416) psect flag "merge" redefined (Assembler)
A psect with a name seen before specifies a different `merge` flag value to that previously seen.

```
psect mytext,class=CODE,merge=1
; and later
psect mytext,class=CODE,merge=0
; Oops, can mytext be merged or not?
```

(1417) argument to "split" psect flag must be 0 or 1 (Assembler)
This psect flag must be assigned a 0 or 1.

```
psect mytext,class=CODE,split=5
```

Oops, the `split` flag argument must be 0 or 1.

(1418) Attempt to read "control" qualified object which is Write-Only (Code Generator)
An attempt was made to read a write-only register.

```
state = OPTION;  // oops -- you cannot read this register
```

(1419) using the configuration file *; you can override this with the environment variable XC_XML (Driver)
This is the compiler configuration file that is selected during compiler setup. This can be changed via the `XC_XML` environment variable. This file is used to determine where the compiler has been installed. See message 1205.

(1420) ignoring suboption "*" (Driver)
The suboption you have specified is not valid in this implementation and will be ignored.

```
--RUNTIME=default,+ramtest
```

oops -- what is ramtest?

(1421) the qualifier __xdata is not supported by this architecture (Parser)
The qualifier you have specified is not valid in this implementation and will be ignored.

```
__xdata int coeff[2];  // that has no meaning for this target
```

(1422) the qualifier __ydata is not supported by this architecture (Parser)
The qualifier you have specified is not valid in this implementation and will be ignored.

```
__ydata int coeff[2];  // that has no meaning for this target
```

(1423) case ranges are not supported (Driver)
The use of GCC-style numerical ranges in case values does not conform to the CCI Standard. Use individual case labels and values to conform.

```
switch(input) {
case 0 ... 5:    // oops -- ranges of values are not supported
    low();
```

(1424) short long integer types are not supported (Parser)
The use of the `short long` type does not conform to the CCI Standard. Use the corresponding long type instead.

```
short long typeMod;    // oops -- not a valid type for CCI
```

(1425) __pack qualifier only applies to structures and structure members (Parser)
The qualifier you have specified only makes sense when used with structures or structure members. It will be ignored.

```
__pack int c;  // oops -- there aren't inter-member spaces to pack in an int
```

(1426) 24-bit floating point types are not supported; * have been changed to 32-bits (Driver)
Floating-point types must be 32-bits wide to conform to the CCI Standard. These types will be compiled as 32-bit wide quantities.

```
-fshort-double=24
```

oops -- you cannot set this `double` size

(1427) machine-dependent path specified in name of included file; use -I instead (Preprocessor)
To conform to the CCI Standard, header file specifications must not contain directory separators.

```
#inlcude <inc\lcd.h>    // oops -- do not indicate directories here
```

Remove the path information and use the `-I` option to indicate this, for example:

```
#include <lcd.h>
```

and issue the `-Ilcd` option.

(1428) "*" is not supported; this feature will be ignored (Driver)
The specified option is not supported and will have no effect on compilation.

```
xc8-cc -mcpu=18f4520 --html main.c
```

Oops, `--html` is not a valid option.

(1429) attribute "*" is not understood by the compiler; this attribute will be ignored (Parser)
The indicated attribute you have used is not valid with this implementation. It will be ignored.

```
int x __attribute__ ((deprecate)) = 0;
```

oops -- did you mean `deprecated`?

(1430) section redefined from “*” to “*” (Parser)
You have attempted to place an object in more than one section.

```
int __section("foo") __section("bar") myvar;  // oops -- which section should it be
in?
```

(1431) the __section specifier is applicable only to variable and function definitions at file-scope (Parser)
You cannot attempt to locate local objects using the `__section()` specifier.

```
int main(void) {
int __section("myData") counter;  // oops -- you cannot specify a section for autos
```

(1432) “*” is not a valid section name (Parser)
The section name specified with `__section()` is not a valid section name. The section name must conform to normal C identifier rules.

```
int __section("28data") counter;  // oops -- name cannot start with digits
```

(1433) function “*” could not be inlined (Assembler)
The specified function could not be made in-line. The function will called in the usual way.

```
int inline getData(int port)   // sorry -- no luck inlining this
{   //...
```

(1434) missing name after pragma “intrinsic” (Parser)
The `intrinsic` pragma needs a function name. This pragma is not needed in most situations. If you mean to in-line a function, see the `inline` keyword or pragma.

```
#pragma intrinsic    // oops -- what function is intrinsically called?
```

(1435) variable “*” is incompatible with other objects in section “*” (Code Generator)
You cannot place variables that have differing startup initializations into the same psect. That is, variables that are cleared at startup and variables that are assigned an initial non-zero value must be in different psects. Similarly, bit objects cannot be mixed with byte objects, like char or int.

```
int __section("myData") input;   // okay
int __section("myData") output;  // okay
int __section("myData") lvl = 0x12;  // oops -- not with uninitialized
bit __section("myData") mode;    // oops again -- no bits with bytes
// each different object to their own new section
```

(1436) “*” is not a valid nibble; use hexadecimal digits only (Parser)
When using `__IDLOC()`, the argument must only consist of hexadecimal digits with no radix specifiers or other characters. Any character which is not a hexadecimal digit will be programmed as a 0 in the corresponding location.

```
__IDLOC(0x51);  // oops -- you cannot use the 0x radix modifier
```

(1437) CMF error * (Cromwell, Linker)
The CMF file being read by Cromwell or the linker is invalid. Unless you have modified or manually generated this file, this is an internal error. Contact Microchip Technical Support with details.

(1438) pragma “*” options ignored (Parser)
You have used unsupported options with a pragma. The options will be ignored.

```
#pragma inline=forced  // oops -- no options allowed with this pragma
```

(1439) message: * (Parser)
This is a programmer generated message; there is a pragma directive causing this advisory to be printed. This is only printed when using IAR C extensions.

```
#pragma message "this is a message from your programmer"
```

(1440) big-endian storage is not supported by this compiler (Parser)
You have specified the `__big_endian` IAR extension for a variable. The big-endian storage format is not supported by this compiler. Remove the specification and ensure that other code does not rely on this endianism.

```
__big_endian int  volume;  // oops -- this won’t be big endian
```

(1441) use __at() instead of '@' and ensure the address is applicable (Parser)
You have used the `@` address specifier when using the IAR C extensions. Any address specified is unlikely to be correct on a new architecture. Review the address in conjunction with your device data sheet. To prevent this warning from appearing again, use the reviewed address with the `__at()` specifier instead.

(1442) type used in definition is incomplete (Parser)
When defining objects, the type must be complete. If you attempt to define an object using an incomplete type, this message is issued.

```
typedef struct foo foo_t;
foo_t x;   // oops -- you cannot use foo_t until it is fully defined
struct foo {
int i;
};
```

(1443) unknown --EXT sub-option “*” (Driver)
The suboption to the `--EXT` option is not valid.

```
xc8 --chip=18f8585 x.c --ext=arm --ext=cci
```

Oops -- valid choices are `iar`, `cci` and `xc8`

(1444) respecified C extension from “*” to “*” (Driver)
The `--EXT` option has been used more than once, with conflicting arguments. The last use of the option will dictate the C extensions accepted by the compiler.

```
xc8 --chip=18f8585 x.c --ext=iar --ext=cci
```

Oops -- which C extension do you mean?

(1445) #advisory: * (Preprocessor)
This is a programmer generated message; there is a directive causing this advisory to be printed.

```
#advisory "please listen to this good advice"
```

(1446) #info: * (Preprocessor)
This is a programmer generated message; there is a directive causing this advisory to be printed. It is identical to `#advisory` messages (1445).

```
#info "the following is for your information only"
```

(1447) extra -L option (-L*) ignored (Preprocessor)
This error relates to a duplicate `-L` option being passed to the preprocessor. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1448) no dependency file type specified with -L option (Preprocessor)
This error relates to a malformed `-L` option being passed to the preprocessor. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1449) unknown dependency file type (*) (Preprocessor)
This error relates to a unknown dependency file format being passed to the preprocessor. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1450) invalid --*-spaces argument (*) (Cromwell)
The option passed to Cromwell does not relate to a valid memory space. The space arguments must be a valid number that represents the space.

```
--data-spaces=a
```

Oops — a is not a valid data space number.

(1451) no * spaces have been defined (Cromwell)
Cromwell must be passed information that indicates the type for each numbered memory space. This is down via the `--code-spaces` and `--data-spaces` options. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

(1452) one or more spaces are defined as data and code (Cromwell)
The options passed to Cromwell indicate memory space is both in the code and data space. Unless you are explicitly running this application, consider this an internal error. Contact Microchip Technical Support with details.

```
--code-space=1,2  --data-space=1
```

Oops — is space 1 code or data?

(1453) stack size specified for non-existent or unused * software stack (Driver)
The `-mstack` option has been used, specifying a maximum size for some or all of the available stacks, but the compiler cannot see that the indicated stack has been used by the program. For example, space might have been requested for an interrupt stack, but there was no corresponding interrupt function found in the code.

```
-mstack=reentrant:20:20:auto
```

Oops, you have asked for two interrupt stacks, but the compiler cannot see both interrupt function definitions.

(1454) stack size specified (*) is greater than available (*) (Driver)
The `-mstack` option has been used to specify the maximum sizes for each stack, but the total amount of memory requested exceeds the amount of memory available.

```
-mstack=software:1000:1000:20000
```

Oops, that is too much stack space for a small device.

(1455) unrecognized stack size “*” in “*” (Driver)
The `-mstack` option has been used to specify the maximum sizes for each stack, but one or more of the sizes are not a valid value. Use only decimal values in this option, or the token auto, for a default size.

```
-mstack=software:30:all:default
```

Oops, only use decimal numbers or `auto`.

(1456) too many stack size specifiers (Driver)
Too many software stack maximum sizes have been specified in the `-mstack` option. The maximum stack sizes are optional. If used, specify one size for each interrupt and one for main-line code.

```
-mstack=reentrant:20:20:auto
```

Oops, too many sizes for a device with only one interrupt.

(1457) local variable "*" cannot be made absolute (Code Generator)
You cannot specify the address of any local variable, whether it be an auto, parameter, or static local object.

```
int pushState(int a) {
int cnt __at(0x100); // oops -- you cannot specify an address ...
```

(1458) Omniscient Code Generation not available in Standard mode (Driver)
This message warns you that not all optimizations are enabled in the Standard operating mode.

(1459) peripheral library support is missing for the * (Driver)
The peripheral libraries do not have code present for the device you have selected. Disable the option that links in the peripheral library.

(1460) function-level profiling is not available for the selected chip (Driver)
Function profiling is only available for PIC18 or enhanced mid-range devices. If you are not using such a device, do not attempt to use function profiling.

(1461) insufficient h/w stack to profile function "*" (Code Generator)
Function profiling requires a level of hardware stack. The entire stack has been used by this program so not all functions can be profiled. The indicated function will not have profiling code embedded into it, and it will not contribute to the profiling information displayed by MPLAB X IDE.

(1462) reentrant data stack model option conflicts with stack management option and will be ignored (Code Generator)
The managed stack option allows conversion of function calls that would exceed the hardware stack depth to calls that will use a lookup table. This option cannot be enabled if the reentrant function model is also enabled. If you attempt to use both the managed stack and reentrant function model options, this message will be generated. Code will be compiled with the stack management option disabled. Either disable the reentrant function model or the managed stack option.

(1463) reentrant data stack model not supported on this device; using compiled stack for data (Code Generator)
The target device does not support reentrant functions. The program will be compiled so that stack-based data is placed on a compiled stack.

(1464) number of arguments passed to function "*" does not match function's prototype (Code Generator)
A function was called with arguments, but the declaration of the function had an empty parameter list (as opposed to a parameter list of void).

```
int test();    // oops--this should define the parameters
...
test(12, input);
```

(1465) the stack frame size for function "*" (* bytes) has exceeded the maximum allowable (* bytes) (Code Generator)
The compiler has been able to determine that the software stack requirements for the named function's auto, parameter, and temporary variables exceed the maximum allowable. The limits are 31 for enhanced mid-range

devices and 127 for PIC18 devices. Reduce the size or number of these variables. Consider `static` local objects instead of auto objects.

```
reentrant int addOffset(int offset) {
int report[400];    // oops--this will never fit on the software stack
```

(1466) registers * unavailable for code generation of this expression (Code Generator)
The compiler has been unable to generate code for this statement. This is essentially a “can’t generate code” error message (message 712), but the reason for this inability to compile relates to there not being enough registers available. See message 712 for suggested workarounds.

(1467) pointer used for writes includes read-only target “*” (Code Generator)
A pointer to a non-`const`-qualified type is being used to write a value, but the compiler knows that this pointer has targets (the first of which is indicated) that have been qualified `const`. This could lead to code failure or other error messages being generated.

```
void keepTotal(char * cp) {
    *cp += total;
}
char c;
const char name[] = "blender";
keepTotal(&c);
keepTotal(&name[2]);  // oops--will write a read-only object
```

(1468) unknown ELF/DWARF specification (*) in --output option (Driver)
The ELF suboption uses flags that are unknown.

```
—output=elf:3
```

Oops, there is no `elf` flag of 3.

This `elf` suboption and its flags are usually issued by the MPLAB X IDE plugin. Contact Microchip Technical Support with details of the compiler and IDE if this error is issued.

(1469) function specifier “reentrant/software” used with “*” ignored (Code Generator)
The `reentrant` (or `software`) specifier was used with a function (indicated) that cannot be encoded to use the software stack. The specifier will be ignored and the function will use the compiled stack.

```
reentrant int main(void) // oops--main cannot be reentrant
{ ...
```

(1470) trigraph sequence “??*” replaced (Preprocessor)
The preprocessor has replaced a trigraph sequence in the source code. Ensure you intended to use a trigraph sequence.

```
char label[] = “What??!”;  // you do know that’s a trigraph
// sequence, right?
```

(1471) indirect function call via a NULL pointer ignored (Code Generator)
The compiler has detected a function pointer with no valid target other than `NULL`. That pointer has been used to call a function. The call will not be made.

```
int (*fp)(int, int);
result = fp(8,10);  // oops--this pointer has not been initialized
```

(1472) --CODEOFFSET option ignored: * (Driver)
The compiler is ignoring an invocation of the `-mcodeoffset` option. The printed description will indicate whether the option is being ignored because the compiler has seen this option previously or the compilation mode does not support its use.

(1474) read-only target “*” may be indirectly written via pointer (Code Generator)
This is the same as message 1467, but for situations where an error is required. The compiler has encountered a pointer that is used to write, and one or more of the pointer’s targets are read-only.

```
const char c = 'x';
char * cp = &c; // will produce warning 359 about address assignment
*cp = 0x44;     // oops--you ignored the warning above, now you are
// actually going to write using the pointer?
```

(1478) initial value for “*” differs to that in *:* (Code Generator)
The named object has been defined more than once and its initial values do not agree. Remember that uninitialized objects of static storage duration are implicitly initialized with the value zero (for all object elements or members, where appropriate).

```
char myArray[5] = { 0 };
// elsewhere
char myArray[5] = {0,2,4,6,8};  // oops--previously initialized
// with zeros, now with different values
```

(1479) EEPROM data not supported by this device (Parser)
The `eeprom` qualifier was used but there is no EEPROM on the target device. Any instances of this qualifier will be ignored.

```
eeprom int serialNo;          // oops--no EEPROM on this device
```

(1480) initial value(s) not supplied in braces; zero assumed (Code Generator)
The assignment operator was used to indicate that the object was to be initialized, but no values were found in the braces. The object will be initialized with the value(s) 0.

```
int xy_map[3][3] = { };   // oops--did you mean to supply values?
```

(1481) call from non-reentrant function, “*”, to “*” might corrupt parameters
(Code Generator)

If several functions can be called indirectly by the same function pointer, they are called ‘buddy’ functions, and the parameters to buddy functions are aligned in memory. This allows the parameters to be loaded without knowing exactly which function was called by the pointer (as is often the case). However, this means that the buddy functions cannot directly or indirectly call each other.

```
// fpa can call any of these, so they are all buddies
int (*fpa[])(int) = { one, two, three };
int one(int x) {
    return three(x+1);  // oops--one() cannot call buddy three()
}
```

(1482) absolute object * overlaps * (Linker)
The reservation for an absolute object has been found to overlap with the memory reserved by another absolute object.

```
unsigned char nfo[6] __at(0x80);
unsigned char nfo2[6] __at(0x7b);  //oops--this overlaps nfo
```

(1483) __pack qualifier ignored (Parser)
The `__pack` qualifier has no affect on auto or `static` local structures and has been ignored.

```
int setInput(void) {
    __pack struct {       //oops--this will not be packed
        unsigned x, y;
    } inputData;
...
```

(1484) the branch errata option is turned on and a BRW instruction was detected (Assembler)
The use of this instruction may cause code failure with the selected device. Check the published errata for your device to see if this restriction is applicable for your device revision. If so, remove this instruction from hand-written assembly code.

```
btfsc status,2
brw  next      ;oops--this instruction cannot be safely used
call  update
```

(1485) * mode is not available with the current license and other modes are not permitted by the NOFALLBACK option (Driver)
This compiler's license does not allow the requested compiler operating mode. Since the `--nofallback` option is enabled, the compiler has produced this error and will not fall back to a lower operating mode. If you believe that you are entitled to use the compiler in the requested mode, this error indicates that your compiler might not be activated correctly.

(1486) size of pointer cannot be determined during preprocessing. Using default size * (Preprocessor)
The preprocessor cannot determine the size of pointer type. Do not use the `sizeof` operator in expressions that need to be evaluated by the preprocessor.

```
#if sizeof(int *) == 3   // oops - you can't take the size of a pointer type
#define MAX 40
#endif
```

(1488) the stack frame size for function "*" may have exceeded the maximum allowable (* bytes) (Code Generator)
This message is emitted in the situation where the indicated function's software-stack data has exceeded the theoretical maximum allowable size. Data outside this stack space will only be accessible by some instructions that could attempt to access it. In some situations the excess data can be retrieved, your code will work as expected, and you can ignore this warning. This is likely if the function calls a reentrant function that returns a large object, like a structure, on the stack. At other times, instructions that are unable to access this data will, in addition to this warning, trigger an error message at the assembly stage of the build process, and you will need to look at reducing the amount of stack data defined by the function.

(1489) unterminated IF directive at end of psect * (Assembler)
The assembler has reached the end of the named psect and not seen the terminating `ENDIF` directive associated with the last `IF` or `ELSIF` directive previously encountered.

```
psect mytext,class=CODE,reloc=2
movlw 20h
IF TEST_ONLY
    movlw 00h
    movwf _mode,c  ; oops--where does the IF end?
psect nexttext,class=CODE,reloc=2
```

(1490) ENDIF not inside an IF directive (Assembler)
The assembler has encountered an `ENDIF` directive that does not have any corresponding `IF` or `ELSIF` directive.

```
psect mytext,class=CODE,reloc=2
movlw 20h
IF TEST_ONLY
    movlw 00h
ENDIF
ENDIF    ; oops--what does this terminate?
```

(1491) runtime sub-option "*" is not available for this device (Driver)
A specified suboption to the `--RUNTIME` option is not available for the selected device.

```
xc8 --CHIP=MCP19114 --RUNTIME=+osccal main.c
```

Oops, the `osccal` suboption is not available for this device.

(1492) using updated 32-bit floating-point libraries; improved accuracy might increase code size (Code Generator)
This advisory message ensures you are aware of the changes in 32-bit floating-point library code operation that might lead to an increase in code size.

(1493) updated 32-bit floating-point routines might trigger "can't find space" messages appearing after updating to this release; consider using the smaller 24-bit floating-point types (Linker)
This advisory message ensures you are aware of the changes in 32-bit floating-point library code operation, which might lead to the Can't Find Space error message that has been issued.

(1494) invalid argument to normalize32 (Assembler)
The `NORMALIZE32` operator has been used on an operand that is not a literal constant.

```
NORMALIZE(_foobar)  ; oops--that must be a literal constant operand
```

(1495) ADDFSR/SUBFSR instruction argument must be 0-3 (Assembler)
The operand to this instruction must be a literal constant and in the range 0 to 3, inclusive.

```
addfsr 1, 6  ; oops--the offset must be between 0 to 3
```

(1496) arithmetic on pointer to void yields Undefined Behavior (Code Generator)
Performing operations on pointers requires the size of the pointed-to object, which is not known in the case of generic (`void *`) pointers.

```
void * vp;
vp++;   // oops—how can this be incremented without knowing what it points to?
```

(1497) more than one *interrupt function defined (Code Generator)
Only one interrupt function of the same priority can be defined.

```
void interrupt lo_isr(void) {  // oops — was this meant to be a low_priority
interrupt?
  …
}
void interrupt hi_isr(void) {
  …
}
```

(1498) pointer (*) in expression may have no targets (Code Generator)
A pointer that contains `NULL` has been dereferenced. Assign the pointer a valid address before doing so.

```
char * cp, c;
c = *cp;    // oops —what is cp pointing to?
```

(1499) only decimal floating-point constants can be suffixed “f” or “F”
The floating-point constant suffix has been used with an integer value.

```
float myFloat = 100f*3.2;  // oops — is ‘100f’ mean to be a hex or floating-point
value?
```

8.4 Messages 1500 Thru 1999

(1500) invalid token in #if expression (Preprocessor)
There is a malformed preprocessor expression.

```
#define LABEL
#define TEST 0
#if (LABEL == TEST) // oops--LABEL has no replacement text
```

(1504) the PIC18 extended instruction set was enabled but is not supported by this compiler (Parser)
The MPLAB XC8 compiler does not support generation of code using the PIC18 extended instruction set. The extended instruction set configuration bit must always be disabled.

```
#pragma config XINST=ON  // oops--this must be disabled at all times
```

(1505) interrupts not supported by this device (Code Generator)
You have attempted to define an interrupt function for a device that does not support interrupts.

```
void interrupt myIsr(void)  // oops--nothing will trigger this
{ ... }
```

(1506) multiple interrupt functions (* and *) defined at interrupt level * (Code Generator)
More than one interrupt function has been defined for the same priority.

```
void interrupt low_priority
isr(void)
{ ... }
void interrupt low_priority    // oops--you can have two ISRs
loisr(void)                 // with the same priority
{ ... }
```

(1507) asmopt state popped when there was no pushed state (Assembler)
The state of the assembler optimizers was popped in assembly code but there was no corresponding push.

```
movlw  20h
movwf  LATB
opt asmopt_pop; oops--there was never a state pushed
```

(1508) specifier “__ram” ignored (Parser)
This pointer-target specifier cannot be used with an ordinary variable and it will be ignored. Confirm that this definition was not meant to indicate a pointer type.

```
__ram int ip;   // oops -- was this meant to be a pointer?
```

(1509) specifier “__rom” ignored (Parser)
This pointer-target specifier cannot be used with an ordinary variable and it will be ignored. Confirm that this definition was not meant to indicate a pointer type.

```
const __rom int cip;   // oops -- was this meant to be a pointer?
```

(1510) non-reentrant function “*” appears in multiple call graphs and has been duplicated by the compiler (Code Generator)
This message indicates that the generated output for a function has been duplicated since it has been called from both main-line and interrupt code. It does not indicate a potential code failure. If you do not want function output duplicated, consider using the hybrid stack model (if possible), or restructure your source code.

(1511) stable/invariant mode optimizations no longer implemented; option will be ignored (Driver)
This option is no longer available and has been ignored.

(1512) stable/invariant mode optimizations no longer implemented; specifier will be ignored (Code Generator)
This specifier is no longer available and has been ignored.

(1513) target “*” of pointer “*” not in the memory space specified by * (Code Generator)
The pointer assigned an address by this statement was defined using a pointer-target specifier. This assignment might be assigning addresses to the pointer that conflict with that memory space specifier.

```
__rom int * ip;
int foobar;
ip = &foobar;  // oops -- foobar is in data memory, not program memory
```

(1514) “__ram” and “__rom” specifiers are mutually exclusive (Parser)
Use of both the `__ram` and `__rom` pointer-target specifiers with the same pointer does not make sense. If a pointer should be able to represent targets in any memory space, do not use either of these specifiers.

```
// oops -- you can’t limit ip to only point to objects in ram and
// also only point to objects in rom
__ram __rom int * ip;
```

(1515) disabling OCG optimizations for this device is not permitted (Driver)
Due to memory limits, projects targeting some devices cannot be built. Ensure that the OCG category of optimization is enabled.

(1516) compiler does not support 64-bit integers on the target architecture
Due to memory restrictions, the current device cannot support 64-bit integers and a smaller integer will be used instead. Choose a type smaller than `long long` to suppress this warning.

```
long long int result;  // oops - this will not be 64-bits wide
```

(1517) peripheral library support only available for C90 (Driver)
The legacy peripheral library was build for the C90 standard and cannot reliably be used for other C standards.

(1518) * function call made with an incomplete prototype (*) (Code Generator)
A function has been called with the compiler not having seen a complete prototype for that function. Check for an empty parameter list in the declaration.

```
void foo();  // oops -- how will this call be encoded?
int main(void)
{
```

```
    foo();
}
```

1519 note-psects will ignore optimisation-related psect flags (Assembler)
Psects using the `note` psect flag cannot be optimized and any additional psect flags which request optimization will be ignored. Psects using the note flag typically contain debug information not related to your project code.

1520 malformed mapfile while generating summary: no space at position * (Driver)
While printing the memory summary after compilation, the psect information read in from the map file was malformed.

1521 internal error encountered creating DWARF information; contact Microchip support with details (Cromwell)
This is an internal compiler error. Contact Microchip Technical Support with details.

1522 RAM access bit operand not specified, assuming * (Assembler)
The assembly instruction is missing the RAM access operand and the assembler has made the stated assumption as to the instruction's destination location. Always use the RAM access bit to ensure the intended operation of your code is clearly stated.

```
    movwf input   ; oops - use for example movwf input,b to indicate banked access,
etc
```

1523 debug_source state popped when there was no pushed state (Assembler)
The state of the `debug_source` setting was popped but no previous value had been pushed.

```
DEBUG_SOURCE asm
    MY_UNLOCK_MACRO
DEBUG_SOURCE pop   ; oops - there was no prior push of the debug source state
```

1524 unrecognized heap size "*" (Driver)
The argument to the `-mheap` option was not a decimal number or valid keyword (for example, `auto`).

```
xc8-cc -mcpu=18f4520 -mheap=0x100 main.c
```

Oops, did you mean to use `-mheap=256` ?

1525 selected language standard or device does not support a memory heap (Driver)
A heap has been specified on a device that does not support dynamic memory allocation, or the C90 language specification was selected with this option. Dynamic memory allocation is only supported with PIC18 or Enhanced Mid-range devices, and only when building with the C99 language standard.

```
xc8-cc -mcpu=12f509 -mheap=0x100 main.c
```

Oops, that device does not support dynamic memory allocation.

1526 total software stack(s) and memory heap size specified (* bytes) is greater than available (* bytes) (Driver)
The size of the memory requested for the software stack and the heap exceeds the amount of data memory available on the selected device.

```
xc8-cc -mcpu=18f4520 main.c -mheap=1000 -mstack=reentrant:4000
```

Oops, that device does not implement 5000 bytes of data memory.

1527 auto-sized software stack(s) and/or memory heap will share in * byte(s) of memory (Driver)
After allocation of static objects, the compiled stack, and any software stacks and/or heap that have a fixed size requested in the `-mstack`/`-mheap` options, any free memory is then divided up for those software stacks and/or

heap that have used an `auto` size specification. If the amount of memory available (as indicated in the message) is very small, this warning indicates that use of the software stack/heap features might be restricted.

1528 no memory can be allocated to software stack(s) and/or memory heap (Driver)
The compiler was unable to allocate memory to any of the software stacks and/or heap.

1529 the -msmart-io-format option has no effect with the currently selected language standard and will be ignored (Driver)
The `-msmart-io-format` option can only be specified when using the C99 language standard, which has full support for smart IO functions.

```
xc8-cc -mcpu=18f4520 main.c -std=c90 -msmart-io-format=fmt="%d%c"
```

Oops, you cannot use the `-msmart-io-format` with the C90 language standard.

1530 insufficient memory available to allocate to * (Driver)
The compiler was unable to allocate memory to the specified software stack/heap.

1600 "*" argument : * (Hexmate)
There is an error in an argument to a Hexmate option. The message indicates the offending argument and the problem.

1601 "*" argument : * (Hexmate)
There is a warning in an argument to a Hexmate option. The message indicates the offending argument and the potential problem.

1602 contents of the hex-data (*) do not conform with the chosen output type (*) (Hexmate)
There is data to be written to the HEX file that is not valid for the particular HEX file format chosen, for example, data might be at an address too large for the supported format. Consider a different output format or check the source of the offending data.

1604 Storage Area Flash has been enabled by the configuration bits; ensure this memory is not used for program code (Assembler)
A configuration bit setting has enabled the storage area flash memory on this device. This memory should not be used by any other part of the program, so it will need to be reserved using an option (e.g. `-mreserve`). Check your device data sheet for more information.

```
#pragma config SAFEN=ON  ;make sure ordinary program code does not use this space
```

1605 Block Table Read Protection has been enabled by the configuration bits; this may affect variable initialization and reading constants in program memory (Assembler)
A configuration bit setting has enabled the block table read protection feature. This can affect any code that reads from program memory, resulting in code failing. Check your device data sheet for more information.

```
#pragma config EBTRB=ON  ;remember this might affect reading program memory
```

8.5 Messages 2000 Thru 2499

(2000) * attribute/specifier has a misplaced keyword (*) (Parser)
An attribute token has been used in a context where it was not expected.

```
// oops -- 'base' is a token which has specific meaning
void __interrupt(irq(base)) isr(void)
```

(2001) * attribute/specifier has a misplaced parenthesis (Parser)
The parentheses used in this attribute construct are not correctly formed. Check to ensure that you do not have extra brackets and that they are in the correct position.

```
void __interrupt(irq((TMR0)) isr(void) // oops -- one too many ’(’s
```

(2002) __interrupt attribute/specifier has conflicting priority-levels (Parser)
More than one priority has been assigned to an interrupt function definition.

```
//oops -- is it meant to be low or high priority?

void __interrupt(irq(TMR0), high_priority, low_priority) tc0Int(void)
```

(2003) * attribute/specifier has a duplicate keyword (*) (Parser)
The same token has been used more than once in this attribute. Check to ensure that one of these was not meant to be something else.

```
//oops -- using high_priority twice has no special meaning

void __interrupt(irq(TMR0), high_priority, high_priority) tc0Int(void)
```

(2004) __interrupt attribute/specifier has an empty “irq” list (Parser)
The `irq()` argument to the `__interrupt()` specifier takes a comma-separated list of interrupt vector numbers or symbols. At least one value or symbol must be present to link this function to the interrupt source.

```
//oops -- irq() does not indicate the interrupt source

void __interrupt(irq(),high_priority) tc0Int(void)
```

(2005) __interrupt attribute/specifier has an empty “base” list (Parser)
The `base()` argument to the `__interrupt()` specifier is optional, but when used it must take a comma-separated list of interrupt vector table addresses. At least one address must be present to position the vector table. If you do not specify the base address with an ISR, its vector will be located in an interrupt vector table located at an address equal to the reset value of the IVTBASE register.

```
//oops -- base() was used but did not indicate a vector table address

void __interrupt(irq(TMR0), base()) tc0Int(void)
```

(2006) __interrupt attribute/specifier has a duplicate “irq” (*) (Parser)
An `irq()` argument to the `__interrupt()` specifier has been used more than once.

```
//oops -- is one of those sources wrong?

void __interrupt(irq(TMR0,TMR0)) tc0Int(void)
```

(2007) __interrupt attribute/specifier has a duplicate “base” (*) (Parser)
The same `base()` argument to the `__interrupt()` specifier has been used more than once.

```
//oops -- is one of those base addresses wrong?

void __interrupt(irq(TMR0), base(0x100,0x100)) tc0Int(void)
```

(2008) unknown “irq” (*) in __interrupt attribute/specifier (Parser)
The interrupt symbol or number used with the `irq()` argument to the `__interrupt()` specifier does not correspond with an interrupt source on this device.

```
//oops -- what interrupt source is TODO?

void __interrupt(irq(TODO),high_priority) tc0Int(void)
```

(2009) * attribute/specifier has a misplaced number (*) (Parser)
A numerical value appears in an attribute where it is not expected.

```
//oops -- this specifier requires specific argument, not a number
void __interrupt(0) isr(void)
```

(2010) __interrupt attribute/specifier contains a misplaced interrupt source name (*) (Parser)
An interrupt source name can only be used as an argument to `irq()`.

```
//oops -- base() needs a vector table address

void __interrupt(irq(TMR0), base(TMR0)) tc0Int(void)
```

(2011) __interrupt attribute/specifier has a base (*) not supported by this device (Parser)
The address specified with the `base()` argument to the `__interrupt()` specifier is not valid for the target device. It cannot, for example, be lower than the reset value of the IVTBASE register.

```
//oops -- the base() address is too low

void __interrupt(irq(TMR0), base(0x00)) tc0Int(void)
```

(2012) * attribute/specifier is only applicable to functions (Parser)
The `__interrupt()` specifier has been used with something that is not a function.

```
// oops -- foobar is an int, not an ISR

__interrupt(irq(TMR0)) int foobar;
```

(2013) argument “*” used by “*” attribute/specifier not supported by this device (Parser)
The argument of the indicated specifier is not valid for the target device.

```
// oops -- base() can’t be used with a device that does not

// support vectored interrupts

void __interrupt(base(0x100)) myMidrangeISR(void)
```

(2014) interrupt vector table @ 0x* already has a default ISR “*” (Code Generator)
You can indicate only one default interrupt function for any vector location not specified in a vector table. If you have specified this twice, check to make sure that you have specified the correct `base()` address for each default.

```
void __interrupt(irq(default), base(0x100)) tc0Int(void) { ...

void __interrupt(irq(default), base(0x100)) tc1Int(void) { ...
// oops -- did you mean to use different different base() addresses?
```

(2015) interrupt vector table @ 0x* already has an ISR (*) to service IRQ * (*) (Parser or Code Generator)
You have specified more than one interrupt function to handle a particular interrupt source in the same vector table.

```
void __interrupt(irq(TMR0), base(0x100)) tc0Int(void) { ...

void __interrupt(irq(TMR0), base(0x100)) tc1Int(void) { ...
// oops -- did you mean to use different different base() addresses?
```

(2016) interrupt function "*" does not service any interrupt sources (Code Generator)
You have defined an interrupt function but did not indicate which interrupt source this function should service. Use the `irq()` argument to indicate the source or sources.

```
//oops -- what interrupt does this service?

void __interrupt(low_priority, base(0x100)) tc0Int(void)
```

(2017) config programming has disabled multi-vectors, "irq" in __interrupt attribute/specifier is ignored (Code Generator)
An interrupt function has used the `irq()` argument to specify an interrupt source, but the vector table has been disabled via the configuration bits. Either re-enable vectored interrupts or use the priority keyword in the `__interrupt()` specifier to indicate the interrupt source.

```
#pragma config MVECEN=0

void __interrupt(irq(TMR0), base(0x100)) tc0Int(void)

// oops -- you cannot disable the vector table then allocate interrupt

// functions a vector source using irq()
```

(2018) interrupt vector table @ 0x* has multiple functions (* and *) defined at interrupt level * (Code Generator)
The program for a device operating in legacy mode has specified a vector table that contains more than one function at the same interrupt priority-level in the same table. In this mode, there can be at most one interrupt function for each priority level in each vector table.

```
#pragma config MVECEN=0

void __interrupt(high_priority) tc0Int(void) {...

void __interrupt(high_priority) tc1Int(void) {...
```

(2019) * interrupt vector in table @ 0x* is unassigned, will be programmed with a * (Code Generator)
In a program for a device operating in legacy mode, an interrupt vector in the indicated vector table has not been programmed with an address. The compiler will program this vector with an address as specified by the `-mundefints` option.

(2020) IRQ * (*) in vector table @ 0x* is unassigned, will be programmed with the address of a * (Code Generator)
The interrupt vector in the indicated vector table has not been programmed with an address. The compiler will program this vector with an address as specified by the `-mundefints` option.

(2021) invalid runtime "*" sub-option argument (*) (Driver)
The argument to a sub-option specified with the `--RUNTIME` option is not valid.

```
--RUNTIME=default,+ivt:reset
```

Oops, the `ivt` suboption requires a numeric address as its argument.

(2022) runtime sub-option “ivt” specifies a base address (0x*) not supported by this device (Driver)
The address specified with the `ivt` sub-option is not valid for the selected target device. It cannot, for example, be lower than the reset value of the IVTBASE register.

(2023) IVT @ 0x* will be selected at startup (Code Generator)
The source code defines more than one IVT and no address was specified with the `ivt` sub-option to the `--RUNTIME` option to indicate which table should be selected at startup. The IVT with the lowest address will be selected by the compiler. It is recommended that you always specify the table address when using this option.

(2024) runtime sub-option “ivt” specifies an interrupt table (@ 0x*) that has not been defined (Driver)
The ivt sub-option to the `--RUNTIME` option was used to specify a IVT address, but this address has not been specified in the source code with any ISR. Check that the address in the option is correct, or check that the `base()` arguments to the `__interrupt()` specifier are specified and are correct.

```
--RUNTIME=+ivt:0x100
```

Oops -- is this the right address? Nothing in the source code uses this base address.

(2025) qualifier * on local variable “*” is not allowed and has been ignored (Parser)
Some qualifiers are not permitted with auto or local static variables. This message indicates that the indicated qualifier has been ignored with the named variable.

```
near int foobar; // oops -- auto variables cannot use near
```

(2026) variables qualified “*” are not supported for this device (Parser)
Some variable qualifiers are not permitted with some devices.

```
eeprom int serialNo; // oops -- can’t use eeprom with PIC18 devices
```

(2027) initialization of absolute variable “*” in * is not supported (Code Generator)
The variable indicated cannot be specified as absolute in the memory space.

```
eeprom char foobar __at(0x40) = 99; // oops - absolute can’t be eeprom
```

(2028) external declaration for identifier “*” doesn't indicate storage location (Code Generator)
The declaration for an external object (e.g., one defined in assembly code) has no storage specifiers to indicate the memory space in which it might reside. Code produced by the compiler which accesses it might fail. Use `const` or a bank specifier as required.

```
extern int tapCounter;  // oops - how does the compiler access this?
```

(2029) a function pointer cannot be used to hold the address of data (Parser)
A function pointer must only hold the addresses of function, not variables or objects.

```
int (*fp)(int);
int foobar;
fp = &foobar;  // oops - a variable’s address cannot be assigned
```

(2030) a data pointer cannot be used to hold the address of a function (Parser)
A data pointer (even a generic `void *` pointer) cannot be used to hold the address of a function.

```
void *gp;
int myFunc(int);
gp = foobar;  // oops - a function's address cannot be assigned
```

(2033) recursively called function might clobber a static register it has allocated in expression (Code Generator)
The compiler has encountered a situation where a register is used by an expression that is defined in a function that is called recursively and that expression is part of a larger expression that requires this same function to be called. The register might be overwritten and the code may fail.

```
unsigned long fib_rec(unsigned long n)
{
    // the temporary result of the LHS call to fib_rec() might
    // store the result in a temp that is clobbered during the RHS
    // call to the same function
    return ((n > 1) ? (fib_rec(n-1) + fib_rec(n-2)) : n);
}
```

(2034) 24-bit floating-point types are not CCI compliant; use 32-bit setting for compliance (Parser)
The CCI does not permit the use of 24-bit floating point types. If you require compliance, use the `-no-short-float` and `-no-short-double` options, which will ensure the IEEE standard 32-bit floating-point type is used for `float` and `double` types.

(2035) use of sizeof() in preprocessor expressions is deprecated; use __SIZEOF_*__ macro to avoid this warning (Preprocessor)
The use of `sizeof()` in expressions that must be evaluated by the preprocess are no longer supported. Preprocessor macros defined by the compiler, such as `__SIZEOF_INT__`, can be used instead. This does not affect the C operator `sizeof()` which can be used in the usual way.

```
#if (sizeof(int) > 2)   // oops -- use (__SIZEOF_INT__ > 2) instead
```

(2036) use of @ is not compliant with CCI, use __at() instead (Parser)
The CCI does not permit the definition of absolute functions and objects that use the `@ address` construct. Instead, place `__at(address)` after the identifier in the definition.

```
int foobar @ 0x100;  // oops -- use __at(0x100) instead
```

(2037) short long integer types are not compliant with CCI (Parser)
The CCI does not permit use of the 3-byte `short long` type. Instead consider an equivalent `long int` type.

```
short long input;  // oops -- consider input to be long when using CCI
```

(2038) use of short long integer types is deprecated; use __int24 or __uint24 to avoid this warning (Parser)
The `short long` type specifiers has been replaced with the more portable `__int24` (replacing `short long`) and `__uint24` (replacing `unsigned short long`) types.

```
short long input; // oops -- use __int24 as the type for input
```

(2039) __int24 integer type is not compliant with CCI (Parser)
The CCI does not permit use of the 3-byte `__int24` type. Instead use the `long int` type.

```
__int24 input;  // oops -- use a long type when using CCI
```

(2040) __uint24 integer type is not compliant with CCI (Parser)
The CCI does not permit use of the 3-byte `__uint24` type. Instead use the `unsigned long int` type.

```
__uint24 input;  // oops -- use an unsigned long type when using CCI
```

(2041) missing argument after "*" (Driver)
The specified option requires an argument, but none was detected on the command line.

```
xc8-cc -mcpu=18f4520 -Wl,-Map main.c
```

Oops, the `-Map` option requires a map filename, e.g. `-Wl,-Map=proj.map`.

(2042) no target device specified; use the -mcpu option to specify a target device (Driver)
The driver was invoked without selecting what chip to build for. Running the driver with the `-mprint-devices` option will display a list of all chips that could be selected to build for.

```
xc8-cc main.c
```

Oops, use the `-mcpu` option to specify the device to build for.

(2043) target device was not recognized (Driver)
The top-level driver was not able to identify the family of device specified with the `-mcpu` option.

```
xc8-cc -mcpu=pic io.c
```

Oops, the device name must be exactly one of those shown by `-mprint-devices`.

(2044) unrecognized option "*" (Driver)
The option specified was not recognized by the top-level driver. The option in question will be passed further down the compiler tool chain, but this may cause errors or unexpected behavior.

(2045) could not find executable "*" (Driver)
The top-level driver was unable to locate the specified compiler tool in the usual locations. Ensure you have not moved files or directories inside the compiler install directory.

(2046) identifier length must be between * and *; using default length * (Driver)
The number of characters specified as the largest significant identifier length is illegal and the default length of 255 has been used.

```
-N=16
```

Oops, the identifier length must be between 32 and 255.

(2047) 24-bit floating point types are not supported when compiling in C99 (Driver)
The float and double types must be 32-bits wide when compiling for the C99 Standard. If you need 24-bit floating-point types, then you might be able to select the C90 compliant libraries (using the -mc90lib option) or you must compile for C90.

```
xc8-cc -mcpu=18f4520 -fshort-double main.c
```

Oops, you cannot use 24-bit `double` types with C99.

(2048) C language extension "*" is not supported and will be ignored (Driver)
The indicated language extension is not supported.

```
xc8-cc -mcpu=16f1937 -mext=iar main.c
```

Oops, that language extension is not supported.

(2049) C99 compliant libraries are currently not available for baseline or mid-range devices, or for enhanced mid-range devices using a reentrant stack; using C90 libraries (Driver)
At present, C99-compliant libraries are not available for all devices. The C90-compliant libraries can be used with these device while still building your source code to the C99 standard. Alternatively, you may choose to build to the C90 standard.

(2050) use of the -mcci option is deprecated; use -mext=cci to avoid this warning (Driver)
Always use the `-mext=cci` option to select the Common C Interface.

(2051) The current license does not permit the selected optimization level* (Driver)
This compiler's license does not allow the requested compiler operating mode.

```
xc8-cc -mcpu=18f4520 -Os main.c
```

Oops, you cannot select level 's' optimizations if this compiler is unlicensed.

(2052) The current license does not permit the selected optimization level and other levels are not permitted by the NOFALLBACK option (Driver)
This compiler's license does not allow the requested compiler operating mode. Since the `--nofallback` option is enabled, the compiler has produced this error and will not fall back to a lower optimization level. If you believe that you are entitled to use the requested optimizations, this error might indicate that your compiler is not be activated correctly.

(2053) function "*" is never called (Code Generator)
The specified inline function has never been called and will not generate code. This message differs to (520) in that the function specified is marked as inline. You may choose to disable this message for all inline functions, but allow message (520) to be issued for all other unused functions.

(2054) the language standard "*" is not supported; using C99 (Driver)
The language standard specified by the `-std` option is not supported by the compiler. The compiler will use the C99 standard instead.

```
xc8-cc -mcpu=12f510 -std=c11 main.c
```

Oops, you cannot select the C11 standard.

(2056) use of the -fmode option is deprecated; use -O to control optimizations and avoid this warning (Driver)
The compiler no longer uses both the mode and optimization selection to fully specify which optimizations are performed. All optimizations are now controllable via the optimization level, which is selectable using the compiler's `-O` option. Unlicensed compilers, however, cannot use all levels.

(2057) The XC8 compiler installation appears to be corrupted. Please reinstall and try again (Driver)
The compiler has detected that something about the installation is not valid. This is most like due to compiler applications being deleted or moved.

(2058) function "*" cannot be inlined with code coverage enabled (Code Generator)
With the code coverage feature enabled, functions cannot be inlined. This advisary is just reminding you that the indicated function will not be inlined while code coverage is still in effect.

(2059) conflicting * register values found in Start Segment Address record (3) (Hexmate)
Hexmate will pass through any type 3 records in the Hex files being processed, but if there is any conflict in the values specified for the CS or IP registers in these records, it will flag this error.

(2060) CRC polynomial unspecified or set to 0 (Hexmate)
If you are calculating a CRC hash value using Hexmate and the polynomial value is zero, this warning will be triggered to indicate that you will be getting a trivial hash result. Typically this will occur if you have forgotten to set the polynomial value in the checksum option.

(2061) word width required when specifying reserve byte order hash (Hexmate)
If you are calculating a CRC reading data words in the Hex file in reverse order, you must specify a word width in bytes with Hexmate's `r` suboption to `-CK`. If you are using the assembler driver, this is specified using the `revword` suboption to `-mchecksum`.

(2062) word width must be * when specifying reserve byte order hash (Hexmate)
If you are calculating a CRC reading data words in the Hex file in reverse order, the word width can only be one of the values indicated in the message. This value is specified with Hexmate's `r` suboption to `-CK`. If you are using the assembler driver, this is specified using the `revword` suboption to `-mchecksum`.

(2063) * address must be a multiple of the word width when performing a reverse byte order hash (Hexmate)
If you are calculating a CRC reading data words in the Hex file in reverse order, the starting and ending addresses must be multiples of the word width specified in Hexmate's `-CK` option or the assembler's `-mchecksum` option.

(2064) PIC18 extended instruction cannot be used when the standard instruction set is selected (Assembler)
PIC18 assembly projects must be set up to use one of the standard or extended instructions sets. This message will be displayed if you have used an extended instruction without having enabled the extended instruction set using the `-misa` option.

(2065) offset out of range (Assembler)
The file register address for this extended PIC18 instruction is out of range.

```
clrf [200]  ; oops
; for extended instructions, the file operand must be less than, for example, 0x60
```

(2066) MESSG directive: * (Assembler)
This is the output message of the `MESSG` assembler directive.

(2067) ERROR directive: * (Assembler)
This is the output of the `ERROR` assembler directive.

(2068) use of the opt control "*" is deprecated; use the corresponding directive (Assembler)
The assembler controls of the form `OPT` *`CONTROL`* should no longer be used. Equivalent directives are available and can be formed by removing the `OPT` token from the control. Instead of using `OPT TITLE "My great project"`, for example, use `TITLE "My great project"`.

(2069) use of the radix option of the list control is deprecated; use the radix directive (Assembler)
The `LIST` assembler control previously allowed the input source to be specified using the `r` argument to the `LIST` option. This should no longer be used. Use the `RADIX` directive instead. Instead of using `OPT LIST r=hex`, for example, use `RADIX hex`.

(2070) device specified by the PROCESSOR directive conflicts with that set by the -mcpu option (Assembler)
The `-mcpu` driver option sets the target device being built for. The `PROCESSOR` directive may be used, if required, to ensure that an assembly source file is only ever built for the specified device. If there is a mismatch in the device specified by the option and the directive, this message will be displayed.

(2071) could not find record containing hash starting address 0x* (Hexmate)
Hexmate was asked to calculate a hash from data starting at an address that did not appear in the HEX file.

(2072) only SHA256 is currently supported (set width control to 256) (Hexmate)
The width suboption can only be set to 256 or -256 when selecting a SHA hash algorithm. Alternatively, the width suboption can be omitted entirely.

(2073) when compiling for C90, specifying an output file for dependencies is not supported and will be ignored (Driver)
Only the Clang front end can create a file containing dependencies.

```
xc8-cc -mcpu=18f4520 -MF depfile -std=c90 main.c
```

Oops, you cannot use the `-MF` option with `-std=c90`.

(2074) word size for byte skip with hash calculation must be * (Hexmate)
The argument to the `s` suboption of `-CK`, which indicates the size of the word in which bytes will be skipped or the purposes of calculating a hash value is not permitted.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s1
```

Oops, the argument to `s` must be larger than 1.

(2075) word size required when requesting byte skip with hash calculation (Hexmate)
An argument to the `s` suboption of `-CK` is required. It represents the word width in which bytes will be skipped for the purposes of calculating a hash value.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s.2
```

Oops, a number is required after the `s`, for example `s4.2`.

(2076) number of bytes for byte skip with hash calculation must be * (Hexmate)
The number of bytes to skip within each word is illegal.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s4.4
```

Oops, the number of bytes to skip must be less than 4, the skip word width, for example `s4.2`.

(2077) number of bytes required when requesting byte skip with hash calculation (Hexmate)
An argument following the `.` in the `s` suboption of `-CK` is required. It represents the number of bytes to skip in each word for the purposes of calculating a hash value.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021s4.
```

Oops, a number is required after the `.` in the `s` argument, for example `s4.2`.

(2078) the number of hash bytes to which the trailing code is appended (*) must be no greater than the hash width (*) (Hexmate)
A trailing code has been requested to follow the specified number of bytes of the hash value, but this number is larger than that the entire hash.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-2p1021t00.4
```

Oops, if the hash value is only 2 bytes long, asking for a trailing code to be appended after every 4 bytes makes no sense. Instead try `t00.1`, for example, to append the code to each byte.

(2079) the hash width (*) must be a multiple of the number of hash bytes appended with a trailing code (*) (Hexmate)
A trailing code has been requested to follow the specified number of bytes of the hash value, but this number is not a multiple of the hash width.

```
hexmate main.hex -ck=0-ff@100+ffffg5w-4p1021t00.3
```

Oops, if the hash value is 4 bytes long, asking for a trailing code to be appended after every 3 bytes makes no sense. Instead try `t00.2`, for example, to append the code to every two bytes of the hash.

(2080) WARN: * (Assembler)
This is a programmer-generated warning; there is an assembly directive causing a deliberate warning. Check the source code to determine if the warning should be investigated further. Consider removing the directive if it is no longer pertinent.

```
WARN "I have not yet confirmed that this is the correct threshold"
movlw 22
movwf threshold,c
```

(2081) the device architecture "*" does not match previously encountered object files "*" (Linker)
The linker was passed objects files that were built for different target devices. This is only likely to occur if you are using precompiled object files or libraries built from assembler code. Ensure all modules are built for the same device.

(2082) can't create temporary file (Driver)
The driver's attempt to create a temporary file failed. This could be due to a number of reasons, including disk space or permissions for the directory in which the system will create temporary files.

(2083) the current license does not permit the "*" feature (Driver)
A feature has been used that is not permitted by the compiler license. A PRO compiler license is required for some compiler features. Ensure that your compiler is installed correctly and has not expired if you believe that you have the appropriate license.

```
xc8-cc -mcpu=16f1937 -mchp-stack-usage main.c perip.c
```

Oops, the stack guidance feature, for example, requires a PRO compiler license to operate.

(2084) absolute object "*" has an address that lies within memory utilized by the compiler and will likely lead to code failure (Code Generator)
An absolute object (one declared using `__at()`) has been placed at an address that must be used by the compiler. The object will likely be corrupted by compiler generated code at runtime and lead to program failure.

```
int __at(0x0) myVar;  // oops - not a good address
```

(2085) the BANKISEL directive has no effect with the currently selected device and will be ignored (Assembler)
The `BANKISEL` assembler directive is required only for Baseline and Mid-range devices. It will be ignored for other devices, but check your device data sheet to ensure your code that performs indirect access of objects is valid for the selected device.

```
  BANKISEL myVar  ;oops - this device does not use separate indirect access bank
bits
  movwf INDF
```

(2086) memory for a heap has been reserved in response to the detection of calls to malloc/calloc/realloc/free function(s) (Driver)
The compiler has detected calls to the standard dynamic memory allocation functions when the `auto` size value was set with the `-mheap` option. A compiler-determined amount of memory has been reserved for the heap.

(2087) memory for the * software stack has been reserved in response to the detection of functions built for reentrancy (Driver)
The compiler has detected functions built with the reentrant (software stack) model when the `auto` size value was set with the `-mstack` option. A compiler-determined amount of memory has been reserved for the indicated stack.

(2088) fixup overflow referencing * * (0x*) into * byte* at 0x*/0x* -> 0x* (* */0x*) (Linker)**
This is the warning variant of message 1356, which might be issued if using the `--fixupoverflow=warn` linker option. See error message 1356 for more information.

(2089) fixup overflow storing 0x* in * byte* at 0x*/0x* -> 0x* (* */0x*) (Linker)**
This is the warning variant of message 1357, which might be issued if using the `--fixupoverflow=warn` linker option. See error message 1356 for more information.

(2090) fixup overflow storing 0x* in * byte* (Linker)
This is the list file warning variant of message 1357, which might be printed in the assembly list file if using the `--fixupoverflow=warn` linker option. See error message 1356 for more information.

(2091) fixup overflow messages have been recorded in the list file "*" (Linker)
The `lstwarn` argument to the `--fixupoverflow` was specified and fixup overflow situations were detected in the program. This advisory message is a reminder to check the assembly list file to see where fixup overflows were detected.

(2092) relocatable list file is not available; fixup overflow list file messages cannot be generated (Linker)
The `lstwarn` argument to the `--fixupoverflow` was specified and fixup overflow situations were detected in the program, but the linker is not processing any assembly list file, so the warning messages cannot be printed. The linker completes the information in the assembly list file produced by the assembler after the link step is complete. Check to ensure that a list file was requested, and that you have not explicitly used the linker's `--norlf` option.

(2093) number of arguments passed to function "*" does not match function's prototype (Code Generator)
This is the error variant of message 1464. A function was called with arguments, but the declaration of the function had an empty parameter list (as opposed to a parameter list of void).

```
int test();    // oops--this should define the parameters
...
test(12, input);
```

The error form of this message is printed if the function in question is not externally defined.

(2094) The selected Device Family Pack (DFP) contains features not implemented in this compiler version; consider using an alternate DFP or a more recent compiler release (Driver, Code Generator, Assembler)
The compiler has detected features in the DFP which it does not implement. This is mostly likely the result of selecting a DFP that is much more recent than the compiler. Choose an older DFP or a newer version of the compiler with the same DFP.

9. Document Revision History

Revision A (March 2020)

- Initial release of this document, based on the assembler chapter from the *MPLAB® XC8 C Compiler User's Guide* (DS50002737).

Revision B (September 2021)

- Added screen captures of the MPLAB X IDE project property dialogs corresponding to the assembler command-line options
- Added information on new `BANKISEL` directive
- Added information on previously undocumented `FILE` and `LINE` assembler directives
- Added description of string support for the `DB`, `DW`, and `DDW` assembler directives
- Updated information relating to the quoting of arguments for the `CONFIG` directive
- Added updated information on Hexmate's handling of INHX16 format HEX files
- Added new `O` suboption to Hexmate's `-CK` option which requests the final hash result be XORed with a the specified value

Revision C (May 2022)

- Added new `--fixupoverflow` linker option that performs automatic masking of instruction operand values, and highlighted the use of this option as an alternative to the use of manual address masking in assembly code examples.
- Expanded information on how to access SFRs from assembly code
- Expanded description of assembler labels
- Added MPLAB X IDE Memory options dialog and description
- Updated MPLAB X IDE Output options dialog and description
- Added description of `-c` linker option to control callgraph
- Added description of entry ranges associated with the `-A` linker option, applicable for Baseline devices
- Added description of `-k` linker option to prevent the overlay of auto/parameter blocks defined in assembly code
- Added dedicated descriptions for the FN-type compiled stack assembler directives
- Added section for the `PUBLIC` directive
- Added a section on the use of long command lines with the `pic-as` assembler driver
- Updated `--edf` option to reflect the new name used for message description file
- Updated error and warning messages to reflect new and changed messages produced by the compiler/assembler
- Removed references to temporary labels, which are not supported

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 devices:

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

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

Legal Notice

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with

your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Trademarks

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

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

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

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

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

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

ISBN: 978-1-6683-0484-6

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