Notice to Customers

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

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

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

Table of Contents

1. Preface

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation. In most cases, formatting conforms to the *OASIS Darwin Information Typing Architecture (DITA) Version 1.3 Part 3: All-Inclusive Edition*, 19 June 2018.

Table 1-1. Documentation Conventions

Description	Implementation	Examples
References	DITA: cite	*Hexmate User's Guide.*
Emphasized text	Italics	...is the *only* compiler...
A window, window pane or dialog name.	DITA: wintitle	the **Output** window. the **New Watch** dialog.
A field name in a window or dialog.	DITA: uicontrol	Select the **Optimizations** option category.
A menu name or item.	DITA: uicontrol	Select the **File** menu and then **Save**.
A menu path.	DITA: menucascade, uicontrol	**File** > **Save**
A tab	DITA: uicontrol	Click the **Power** tab.
A software button.	DITA: uicontrol	Click the **OK** button.
A key on the keyboard.	DITA: uicontrol	Press the **F1** key.
File names and paths.	DITA: filepath	`C:/Users/User1/Projects`
Source code: inline.	DITA: codeph	Remember to `#define START` at the beginning of your code.

Source code: block.	DITA: codeblock	An example is:
		`#include <xc.h>`
		`main(void) {`
		`  while(1);`
		`}`

User-entered data.	DITA: userinput	Type in a device name, for example `PIC18F47Q10`.
Keywords	DITA: codeph	`static`, `auto`, `extern`
Command-line options.	DITA: codeph	`-Opa+, -Opa-`
Bit values	DITA: codeph	`0, 1`
Constants	DITA: codeph	`0xFF, 'A'`
A variable argument.	DITA: codeph + option	*`file`*`.o`, where *`file`* can be any valid file name.
Optional arguments	Square brackets [ ]	`xc8 [`*`options`*`] `*`files`*
Choice of mutually exclusive arguments; an OR selection.	Curly brackets and pipe character: { | }	`errorlevel {0|1}`
Replaces repeated text.	Ellipses...	*`var_name`* `[, `*`var_name`*`...]`

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

A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	N'Rnnnn	4'b0010, 2'hF1
Device Dependent insignia. Specifies that a feature is not supported on all devices. Please see your device data sheet for details.	[DD]	Assembler Special Operators

1.2 Recommended Reading

This user's guide describes the use and features of the Hexmate application. The following Microchip documents are available and recommended as supplemental reference resources.

MPLAB® XC8 C Compiler User's Guide

The Hexmate application is shipped with and automatically executed by the MPLAB XC8 C Compiler. If you are using this compiler to build projects, you can get access to most Hexmate features directly from the compiler. This user's guide indicates the compiler options that will invoke Hexmate functions.

MPLAB® XC8 C Compiler Release Notes for PIC® MCU

Changes made to Hexmate are listed in the MPLAB® XC8 C Compiler Release Notes (an HTML file) in the Docs subdirectory of the compiler's installation directory. The release notes contain update information and known issues that cannot be included in this user's guide.

Development Tools Release Notes

For the latest information on using other development tools, refer to the tool-specific Readme files in the docs subdirectory of the MPLAB X IDE installation directory.

2. Hexmate

The Hexmate application is a post-link utility that can merge multiple Intel HEX files into one, reformat HEX files, as well as insert useful data into these files after they have been generated by a compiler.

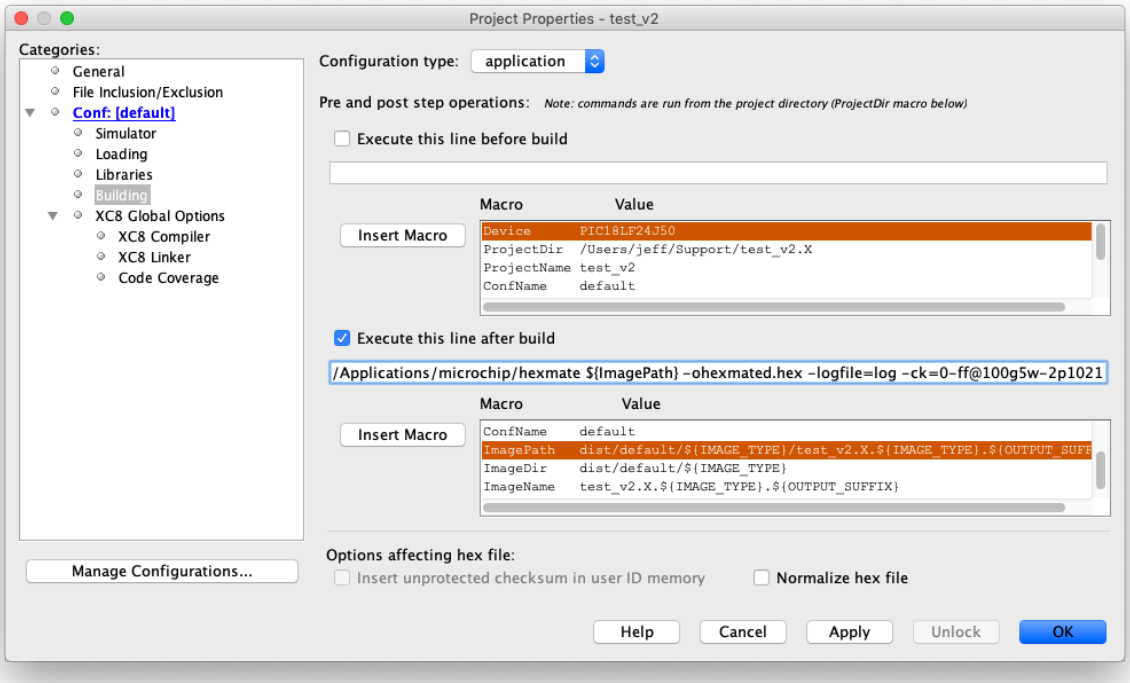
Hexmate is shipped with the MPLAB XC8 C Compiler, but it can be executed as a stand-alone application to process Intel HEX files produced by *any* compiler. The MPLAB X IDE also ships with its own copy of Hexmate, which is used to merge HEX files when building loadable projects.

This guide has information on the tasks that Hexmate can perform and the command-line options that control the application.

When building MPLAB XC8 projects, some Hexmate features can be accessed directly using options to the `xc8-cc` command line driver or by selecting widgets in the MPLAB X IDE **Project Properties** dialog; however, less commonly used features require Hexmate to be called explicitly after the project is built. If your project is built by another compiler, Hexmate must always be executed explicitly to process the HEX file generated from that compiler.

Hexmate can be added as a post-link build command inside the MPLAB X IDE, so you do not need access to a terminal application to run it. In the following screen shot of the **Project Properties** dialog in the MPLAB X IDE, Hexmate has been configured in a post-build step to calculate a CRC over data in the HEX file produced by the project. The *MPLAB® X IDE User's Guide* has further information on how to set project build properties.

Figure 2-1. Executing Hexmate as a post-build step in the MPLAB X IDE



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 Hex files to other INHX formats (e.g., INHX32).
- 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 standardizing the length of data records in a HEX file.
- Validating record checksums within Intel HEX files.

Typical applications for Hexmate might include:

- Merging a bootloader or debug module into a main application at build time.
- Creating a hash value that can be used to ensure programmed code has not become corrupted.
- Placing instructions at unused locations to handle the wayward execution of crashed programs.
- Storing serial numbers or strings such as time stamps at a fixed address.
- Storing initial values at a particular memory address (e.g., initialize EEPROM).
- Detecting the occurrence of a buggy or restricted instructions.
- Adjusting HEX files to meet the particular requirements of bootloaders or debug tools.

2.2 Intel HEX file Specification

Hexmate reads and processes HEX files, in particular, those files conforming to the Hexadecimal Object File Format developed by Intel®. Other forms of HEX files and other file formats cannot be processed by Hexmate. The information in this section describes the HEX file specification, as interpreted by Hexmate when processing input files.

Intel HEX files store binary program data using ASCII characters. One character is used to store each nibble of the binary data, so, for example, the byte value 0xFE would be stored in a HEX file as the ASCII byte 0x46 (representing the most significant nibble of the value, `F`) followed by 0x45 (`E`). If you view a HEX file with a text editor, it will show these two bytes as the characters `FE`, since it assumes the data is ASCII and maps it accordingly for display. Upper case hexadecimal letters are typically used in HEX files, but Hexmate will accept either case.

A HEX file consists of a number of records, with at most one record per line.

A record begins with a Record Mark, that being a colon (`:`) character. Any other characters encountered at the beginning of a line are interpreted by Hexmate as a comment until either a colon or new-line character is encountered. A line in a HEX file is valid if it is:

- entirely a valid record (described below)
- blank (only contains a new-line character)
- entirely a comment (contains any characters excluding those that define a valid record)
- a comment followed by a valid record

Once a Record Mark has been encountered in a line, it and all the following characters on that line are assumed to be part of a record. The record will be checked for syntactic correctness and any errors will be reported, but with one exception: Should a record following a comment be malformed, then the whole line will be treated as a comment and no errors will be reported.

The general form of a record, showing the different fields, is as follows.

Record Mark	Data Length	Address Offset	Record Type	Data/Argument	Checksum
:	*LL*	*OOOO*	*TT*	*DDDD...DDDD*	*CC*

The Data Length field follows the Record Mark and is a one-byte (two ASCII character) value indicating the length of the Data/Argument field in the record. This specification is required, since the length of the Data/Argument field is not fixed, and it can vary from one record to the next.

A two-byte (four character) Address Offset field follows. If a record is one that contains program data, the value in this field (and potentially information contained in other records) indicates the address at which to start programming the data in the device; otherwise, it is not used and contains the character sequence `0000`. The correct Address Offset must be specified with each record that holds data; the location for data cannot "flow on" from any previous record in the file. The mandatory inclusion of an address with each record allows records to be placed in the file in any order, subject to some limitations, described later.

Note that the Address Offset field, as its name implies, specifies an address *offset*, not an absolute address. The base address to which the offset applies is specified by special record types, discussed later in this section; however Hexmate assumes that this base address is 0 if no such special records have been previously encountered in the file.

A single byte (two characters) Record Type value follows the Address Offset and is used to indicate what sort of information the record contains. There are six types of record, tabulated below.

Record Type Characters	Record Name	Purpose
`00`	Data Record	Contains data bytes to be programmed into the device.
`01`	End-of-file Record	Indicates the final record in the file.
`02`	Extended Segment Address Record	Contains the Upper Segment Base Address.
`03`	Start Segment Address Record	Contains the execution start address when using Extended Segment Address Record data.
`04`	Extended Linear Address Record	Contains the Linear Base Address.
`05`	Start Linear Address Record	Contains the execution start address when using Linear Address Record data.

A multipurpose, variable-length Data/Argument field follows the Record Type field for most record types. This field is used to store the data to be programmed for type 0 records, store a base address for type 2 and 4 records, and store a start address for type 3 and 5 records. This field is not present for type 1 records. For type 0 Data Records, the length of the Data/Argument field is permitted to range from 1 to 255 bytes (2 to 510 characters), and this length can vary from one Data Record to the next. The bytes of data in this field assume consecutive addresses, indexed from the first byte, which has the Address Offset specified in the record.

A Checksum field makes up the last byte (2 characters) in the record for all record types and this value is used by Hexmate to minimize the chance of corrupted data being programmed. Record checksums are calculated as an 8-bit two's complement of the summation of each byte (in its binary form) in that record, starting from the Record Length byte to the last byte in the Data/Argument Field.

The following line (formatted to highlight the different fields) is an example of a valid record that might be contained in a HEX file.

```
:04000000FEEFFFF020
```

This line begins with a colon (`:`), signaling the start of a record. There is no comment present on the line. The next two characters (`04`) indicate the length of the Data/Argument field in this record to be 4 bytes (8 characters). The following 4 bolded characters (**0000**) form the Address Offset, that being the value 0. This is followed by two characters (`00`) being the record type, indicating that this is a Data Record containing bytes to be programmed. The following 8 characters of data, `FEEFFFF0`, are underlined. These represent the bytes of data 0xFE, 0xEF, 0xFF, and 0xF0. These bytes have address offsets of 0 through 3, respectively. Finally, there are two characters representing the checksum

value, 20. This value is obtained by first summing the bytes 0x04, 0x00, 0x00, 0x00, 0xFE, 0xEF, 0xFF, and 0xF0 contained in the record, which yields the value 0x3E0. The 8-bit two's complement of this value is 0x20.

Another example including a comment, two Data Records, and one End-of-file Record, forming a complete HEX file is shown below. For illustrative purposes, the Address Offset in each record has been bolded and the Data/Argument field (where present) is underlined.

```
;configured for basic mode
:04000000FDEF0FF011
:0C1FF400B88100EF00F00001FAEF0FF0E0
:0000000001FF
```

Note that HEX files always use byte addresses, that is, each byte of data in each record has a unique address. Some devices use word addressing, where each unique device address might contain more than one byte of data. The program memory addressing size of Microchip device families is shown in the following table.

Device family	Program memory addressing size
8-bit Baseline, Mid-range, and Enhanced Mid-range PIC® MCUs	2 bytes (word)
PIC18 MCUs	1 byte
8-bit AVR® MCUs	2 bytes (word)
PIC24 MCUs	2 bytes (word)
dsPIC® DSCs with 24-bit instruction sets	2 bytes (word)
dsPIC DSCs with 32-bit instruction sets	1 byte
PIC32 and SAM MCUs and MPUs	1 byte

For those Microchip devices that use word addressing, two bytes of data (with two unique HEX file addresses) are needed to program one address location on the device, thus, the HEX file address for a particular byte of data will not be the same as the address at which that byte will be programmed in the device. Always be mindful that this mapping might need to be performed if you are searching a HEX file for data at a particular device address. Hexmate's `-addressing` option (see Addressing Hexmate Option) can assist with this mapping when specifying options.

The following table shows an example of how word-addressed device memory will be programmed from a HEX file record (assuming the base address is 0):

```
:0401FC00E040FEFFE2
```

Table 2-1. Mapping from HEX file address to device address for word-addressed devices

HEX file address	HEX file data	Device Address	Device Data
0x1FC	0xE0	0xFE	0x40E0
0x1FD	0x40		
0x1FE	0xFE	0xFF	0xFFFE
0x1FF	0xFF		

2.2.1 HEX File Formats

Although there is only one Intel HEX file specification, Intel HEX files assume different formats based on the types of records that they contain. These formats were originally developed to accommodate Intel devices with more than 64KB of memory, but they have been adapted to suit other devices. The formats applicable to Microchip tools are described in this section.

If the data to be programmed into a device does not need to be placed at HEX file addresses greater than 0xFFFF (remember that these might map to lower device addresses if the device's memory is word addressable), the HEX file to hold such a program image only needs to contain Data (type 0) and End-of-file (type 1) record types. Any HEX file that contains only these record types is often said to conform to the Microchip INHX8M format.

If data has to be placed at higher locations, Microchip tools use the type 4, Extended Linear Address Record, to allow higher addresses to be used in the HEX file. This record defines the base address to which the Address Offset specified by subsequent Data Records is added. The base address is calculated as the type 4 record's Argument payload shifted left 16 bits. HEX files that use no record types other than type 4 (and the accompanying type 5) records in addition to any type 0 and 1 records are considered to conform to the Microchip INHX32 format.

The following shows how a type 4 record is used in an INHX32-conforming HEX file.

```
:0A00D000 10EEECF2110057EF00F003
:02000004 0030CA
:02000200 FFFFFE
:00000001FF
```

The first line shows a Data Record holding data at HEX-file address offset **0x00D0** (bolded). If this is the first record in the file, then Hexmate will assume that the base address has a default value of 0. The first byte (index 0) in this record (0x10) has HEX file address 00. The next byte of data (index 1) has the value 0xEE and this is located at HEX file address 1, etc.

The following line in the file contains an Extended Linear Address Record, as indicated by the `04` present in the Record Type field. The Data/Argument field for this record holds the value 0x0030 (underlined), which specifies a base address of 0x300000 when shifted left 16 bits. This new base address must be added to the Address Offsets of all subsequent Data Records, unless another type 4 record is used to change that base address. The following Data Record in the file indicates an Address Offset of 0x0002. The data address for the first byte in this record (the byte 0xFF) is calculated as the base address, 0x300000, plus the Address Offset specified by the Data Record, all modulo 2^{32}. So the first data byte has HEX file address 0x300002. The subsequent byte of data in this record (another 0xFF byte) has the address 0x300003.

The type 5 Start Linear Address Record that can accompany type 4 Extended Linear Address Records is typically not used by any Microchip debugging tools; however GCC-based MPLAB XC compilers might produce these records. Hexmate will process these records as described in Sla Hexmate Option, which also describes how this option can be used to insert a type 5 record into the HEX file output.

Hexmate can also read HEX files that are said to conform to the Microchip INHX16 format, although this format is less commonly used. In addition to the type 0 and 1 records, these file formats can also contain type 2 Extended Segment Address Records. Like type 4 records, these specify a base address to which the Address Offset specified by subsequent Data Records is added. The base address is calculated as the Argument payload of the type 2 record shifted left 4 bits.

The following HEX file shows how a type 2 record is used.

```
:0240F0001200BC
:02000002 1000EC
:04FFFC005BEF20F0A7
:00000001FF
```

The first line shows a Data Record holding data at HEX-file address offset **0x40F0** (bolded). If this is the first record in the file, then Hexmate will assume that the base address has a default value of 0. The first byte (index 0) in this record (0x12) has HEX file address 00. The next byte of data (index 1) has the value 0x00 and the HEX file address 1, etc. The following line in the file contains an Extended Segment Address Record, as indicated by the 02 present in the Record Type field. The Argument field for this record holds the value 0x1000 (underlined), which specifies a base address of 0x10000 when shifted left 4 bits, and which must be added to the address of all subsequent Data Records. The following Data Record specifies an address offset of 0xFFFC. The HEX file address for the first byte in this record is calculated as the base address, 0x10000, plus the modulo 2^{16} result of the offset specified by the Data record plus the byte's index in that record. So the first byte of data is located at address 0x1FFFC.

The type 3 Start Segment Address Record that can accompany type 2 Extended Segment Address Records is typically not used by any Microchip debugging tools; however GCC-based MPLAB XC compilers might produce these records. Hexmate will process these records as described in Ssa Hexmate option, which also describes how this option can be used to insert a type 3 record into the HEX file output.

A summary of the record types allowable in each format is tabulated below.

Table 2-2. HEX file Formats

Format	Valid record types	HEX file address range
INHX8M	0, 1	16 bit
INHX16	0, 1, 2, 3	20 bit
INHX32	0, 1, 4, 5	32 bit

If a HEX file contains type 2 or 3 and also contains type 4 or 5 records, then the file does not conform to any named format and these files are often considered invalid. Hexmate *will* process such files, but will always write output files with a recognized format. If a mixed-format file is passed to Hexmate, any type 3 or 5 records not permitted in the output format specified by the `-format` option will be removed; any type 2 record will be converted to a type 4 record when producing an INHX32 format, and any type 4 record will be converted to a type 2 record if possible when producing a INHX16 format; otherwise, an error will be produced.

Note that converting from INHX8M to INHX16, or from either INHX8M or INHX16 to INHX32 can be easily performed. Converting the other way might not be possible. An INHX32 format can only be converted to a INHX16 or INHX8M format if the base addresses specified in the type 4 records do not place data outside the HEX file address range, indicated in the above table, that the format can represent.

2.2.2 Output HEX files

The HEX files output by Hexmate conform to the same specification described for the input files; however they are standardized in a number of ways, which are detailed in this section.

Hexmate will always write output files:

- Using upper case hexadecimal letters, even if lower cases hexadecimal letters were used in the input file.
- Without comments, regardless of whether any comments in the input files were on a line preceding a record or on a line by themselves.
- With output records in ascending Address Offset order following their corresponding base address (type 2 or type 4) record.
- With data destined for contiguous addresses placed into as many records as possible with the Data Length specified by the `-format` option if that option has been used or a length of 16 otherwise, then output the remainder of the data into a record with a smaller length.

The following shows how an input HEX file conforming to the INHX16 format might be modified by Hexmate using a command similar to the following, which has Hexmate read in the file called `input.hex` and output it to the file called `output.hex`, ensuring it conforms to the INHX32 format.

```
hexmate input.hex -format=inhx32 -ooutput.hex
```

Line #	Input HEX file	Output HEX file
1	`%% production file`	`:04000000FEEFFFF020`
2	`begin:04000000feefffff020`	`:1040100061207374726 96E670000000048656C6C03`
3	`:1040800050EF20F005C501F506C502F5000E046EDF`	`:1040800050EF20F005C501F506C502F5000E046EDF`
4	`:1040100061207374726 96E670000000048656C6C03`	`:1040F000120074686572652C204920616D205BEFA9`

.....continued

Line #	Input HEX file	Output HEX file
5	:0240F0001200BC	:0241000020F0AD
6	:0C40F20074686572652C204920616D2007	:020000040001F9
7	:0440FE005BEF20F064	:104000000068692074686572652C204920616D2004
8	:020000021000EC	:00000001FF
9	:104000000068692074686572652C204920616D2004	
10		
11	:00000001FF	

Notice the following differences between the input and output files.

- The whole-line comment on line 1 and the empty line (line 10) of the input file were not output.
- The in-line comment before the record on line 2 of the input was not output (the record it preceded, now appears on line 1 of the output).
- The lowercase hexadecimal letters in the record on line 2 of the input were converted to uppercase letters, as seen in line 1 of the output, where that record now appears.
- The order of lines 3 and 4 of the input was swapped to lines 3 and 2 respectively in the output, so they appear in ascending Address Offset order.
- The data contained in the records appearing on lines 3, 4, and 5 of the input was merged into one record on line 4 of the output with a length of 16 data bytes (the default length) and the remainder of the data was written into another smaller record on line 5.
- The type 2 record on line 8 of the input was replaced with a type 4 record, on line 6 of the output, and its argument value (0x1000) was converted to 0x0001, so that it specifies the same base address.

2.3 Internal Operation

Hexmate can perform many operations on the HEX file(s) it is processing. An appreciation of the order in which these operations are performed is important, as it can affect the output.

The following Hexmate operations are listed in the order in which they are performed.

- Create an internal master image by merging the data contained in all the input HEX files, processing them in the order in which they appear on the command-line and as per the read specification associated with each file.
- Find values specified using `-find` options in the master image, deleting and replacing these values as requested in the reverse order in which the options were issued.
- Insert serial data specified using `-serial` options into the master image in the reverse order in which the options were issued.
- Embed strings specified using `-string` and `-strpack` options into the master image in the reverse order in which the options were issued.
- Reserve space for hashes requested using `-ck` options and trailing codes (`t` argument to `-ck`) in the master image.
- Fill unused locations specified using `-fill` options in the master image in the order in which the options were issued.
- Calculate each specified hash from the master image and insert these and any trailing codes into the master image in the order in which the options were issued.
- Encode the final master image into the Intel HEX file format requested by the `-format` option or a suitable format when this option has not been used, and then output this file as dictated by the `-o` option.

Masking of data specified using `-mask` options is performed whenever the master image is modified, whether the written data has come from an input HEX file or the result of a Hexmate operation being stored in the master image.

2.4 Potential Causes of Failure

Situations where the execution of Hexmate might fail, or where the generated HEX file might lead to runtime failure are indicated in this section. This is typically the result of incorrect configuration of project code, or incorrect compiler or Hexmate options.

The following are typical causes of the data overwrite error, `(944) data conflict at address *h between * and *`, which indicates that more than one record holds data for the same address.

- When writing bootloader/application projects, the configuration words have been specified differently in the application and bootloader projects. Typically, you should specify them in only one project.
- When writing bootloader/application projects, program memory has not been partitioned between the two projects during compilation. Use a compiler option to ensure that the memory used by one project is not used by the other.
- When writing bootloader/application projects, one project defines functions at an absolute address (using `__at()` or `address` attribute) that is within the range of program memory used by the other project. Choose different addresses or avoid using absolute functions.
- When writing XC32 bootloader/application projects, the debug exception handler has not been discarded from the linker scripts, for example, using the following:

```
SECTION
{
  /DISCARD/ : { *(._debug_exception) }
}
```

- A hash, serial number, or string has been embedded at a location used by executable code or other data. Choose a different location.
- A Hexmate-generated hash value has been stored into a region of memory used to calculate the hash itself. For example, a hash calculated over the data between addresses 0-0xFF is stored at location 0x7E. Either change the range of values used to generate the hash, or move the hash result to a new location.

The following are common reasons why hash values (such as checksums, SHAs, etc.) generated by Hexmate do not agree with hashes calculated at runtime.

- The byte order of data within words is processed in different orders, for example, MSB first versus LSB first. Change the algorithm in your code, or specify a *`revWidth`* value in the `-CK` Hexmate option.
- Unused memory space that is included in the hash calculation was not filled with a known value. Use your compiler's or Hexmate's fill option, or adjust the range of addresses used to calculate the hash.
- Software breakpoints have been set when running code on the programmed device. The instructions at these locations are replaced with some form of break instruction by the MPLAB X IDE. If the opcodes of these break instructions are read to calculate a hash value at runtime, that hash value will be different from one calculated at compile time using the usual opcode values that reside at these locations in the HEX file.

The following is a common cause of failure for a number of Hexmate options.

- Device addresses were used with Hexmate options, but the wrong addressing value was specified for the device. See the `-addressing` option (Addressing Hexmate Option) for the addressing value that should be used with your device if you intend to use device addresses with any options.

3. 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 is a space-separated list of input Intel HEX files to merge using Hexmate. If only one HEX file is specified, no merging takes place, but other actions can be performed on the HEX file, as specified by *options*. Options can appear anywhere on the command line and are tabulated below.

Table 3-1. Hexmate Command-line Options

Option (links to explanatory section)	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.
--werror	Promote warnings to errors.
-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 INHX variant for the output and maximum data record length.
-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.
-o*file*	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.
-w*level*	Adjust warning sensitivity.
+ override prefix	Prefix to any option to overwrite other data in its address range, if necessary.

Hexmate can read and write common Intel HEX file formats, which contain only specific subsets of record types. The formats are discussed in HEX File Formats and the Format Hexmate Option specifies which format should be written.

The format or assumed radix of numerical option arguments are detailed with each option description. Unless otherwise indicated, any address specified with these options are to be entered as HEX file addresses, unless you use the Addressing Hexmate Option. HEX file addresses are discussed in Intel HEX file Specification.

3.1 Specifications and Filenames

Hexmate can process Intel HEX files that conform to a variety of formats, as discussed in HEX File Formats. Hex files typically use a `.hex` extension; however this is not mandatory when using Hexmate.

Additional specifications can be applied to each HEX file listed on the command line to place conditions on how this file should be processed. If any specifications are used, they must precede the name of the file they correspond to, using a comma to separate them from the filename.

A range restriction can be applied with the specification `r`*`Start`*`-`*`End`*, where *`Start`* and *`End`* are both assumed to be hexadecimal addresses, whose interpretation is based on the addressing value (see Addressing Hexmate Option). Hexmate will only process data within the address range restriction. For example, assuming that the addressing value is 1, the specification:

```
r100-1FF,myfile.hex
```

will tell Hexmate to use `myfile.hex` as an input file but that it should only process data from that file that has HEX file addresses within the range 0x100-1FF (inclusive).

An address shift can be applied with the specification `s`*`Offset`*, 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 HEX file address in the output file. The offset can be either positive or negative. For example, the specification:

```
r100-1FFs2000,myfile.hex
```

will shift the block of data from 0x100-1FF in `myfile.hex` to the new HEX file address range 0x2100-21FF in the output.

Be careful when shifting data containing executable code. Program code should only be shifted if it is position independent.

3.2 Override Prefix

When the `+` prefix 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, silently overwriting any other data existing at the same addresses. Without this prefix, Hexmate will issue an error if two sources try to store differing data at the same location.

Important: Use this prefix with care, as it suppresses any warnings concerning data overwrite, which might lead to code failure.

For example, if `input.hex` contains data at address 0x1000, the command (`+` used with the `-string` option):

```
hexmate  input.hex  +-string@1000="My string"
```

will have the string specified by the `-string` option placed at address 0x1000 in the output file and the data at the same address in the input file will not appear in the output; however, if you were to use the command (no use of the `+` prefix):

```
hexmate  input.hex  -string@1000="My string"
```

an error would be triggered, alerting you to the data conflict.

The override prefix can also be used with files to indicate that one file should take precedence over another should they contain conflicting data at the same address. So for example if two files were being merged, you could use the command:

```
hexmate  +base.hex  auxillary_data.hex
```

to indicate that the content of `base.hex` should take priority.

3.3 Edf Hexmate Option

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 *`build`*`_en.msgs`. A build date (*`yyyymmddhhmmss`* format) will appear in the file's name and may help you identify more up to date files.

3.4 Emax Hexmate Option

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.

3.5 Msgdisable Hexmate Option

The `--msgdisable=`*`num`* option disables error, warning or advisory messages during execution of Hexmate.

Attention: Use this option with caution, as it might mask problems in input files or with options and lead to code failure.

The option is passed a comma-separated list of message numbers that are to be disabled. See a list of messages and their corresponding number in the *Error and Warning Messages* section towards the end of this document. Any error message numbers in this list are ignored unless they are followed by an `:off` argument. For example:

```
--msgdisable=964,1601
```

If the message list is specified as `0`, then all warnings are disabled.

3.6 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.

3.7 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, only one SSA record 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 discrepancies 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.

3.8 Ver Hexmate Option

The `--ver` option will have Hexmate print version and build information and then quit.

3.9 Werror Option

The `--werror` option can be used to promote Hexmate warnings into errors. As with any other error, these promoted messages will prevent the application from completing.

The `--werror` form of this option promotes all Hexmate warning messages issued into errors. This option might be useful in applications where warnings are not permitted for functional safety reasons.

The `--werror=num` form of this option promotes only the warning identified by *num* to an error. See a list of messages and their corresponding number in the *Error and Warning Messages* section towards the end of this document. If the message indicated by *num* is not a warning, then this option has no effect. Only one message number may be specified with this option; however, you may use this option as many times as required.

The `--wno-error=num` form of this option ensures that the warning identified by *num* is never promoted to an error, even if the `--werror` form of this option is in effect.

3.10 Addressing Hexmate Option

The `-addressing=units` option specifies the addressing units of any addresses used in Hexmate's command line options.

If this option is not used, *units* defaults to the value 1 and all addresses specified with Hexmate options are assumed to be byte addresses, that is, each address specifies one byte of data. This matches the addressing used by Intel HEX files. For example, with the addressing units set to the value 1, the option `-mask=0F@0-FF` will mask any data from HEX file address 0x0 to address 0xFF (inclusive). If the target device is also byte addressable, then this data will be found in the device at the same addresses.

Some device architectures, however, use a native addressing format other than byte addressing. Devices that use 16-bit word addressing, for example, require two uniquely addressed bytes from the HEX file to program a single program memory address, thus there is a mismatch between addresses at which data will be found in the HEX file and the addresses at which that same data will be programmed into the device.

If you prefer to use device addresses with Hexmate options and the device's program memory is not byte addressable, use the `-addressing` option to specify the mapping between the two address spaces. The argument can be a value from 1 to 4 and indicates the number of HEX file bytes that will be stored at each device address. Use of this option is not mandatory, even with word-addressed devices; however, if it is not used, you must remember that any addresses specified with options refer to HEX file addresses, not device addresses.

The following table shows the *`units`* argument that will allow device addresses to be used with Hexmate options, based on the target Microchip device family.

Device family	Option argument
8-bit Baseline, Mid-range, and Enhanced Mid-range PIC MCUs	2
PIC18 MCUs	1 (default value)
8-bit AVR MCUs	2
PIC24 MCUs	2
dsPIC DSCs with 24-bit instruction sets	2
PIC32 and SAM MCUs and MPUs	1 (default value)

For example, if you were using a Mid-range PIC device and you used both the `-addressing=2` and `-mask=0F@0-FF` options, the mask would be performed over the HEX file data that will ultimately be programmed between addresses 0x0 to 0xFF on the device, but in the HEX file, this would be the data located from HEX file address 0x0 to address 0x1FF.

3.11 Break Hexmate Option

The `-break` option ensures that the byte of data at a particular address is placed at the beginning of a data record.

This option takes a comma-separated list of unqualified hexadecimal addresses, whose interpretation is based on the addressing value (see Addressing Hexmate Option). If data would ordinarily be written to a record whose address range crossed any of these listed addresses, this option will limit that record to data up to but excluding the relevant address, and a new data record will be output for the subsequent data.

For example, if Hexmate would normally output data in records that specify addresses 0 through to 0x2F, such as:

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:10001000ADDEADDEADDEADDEADDEADDEADDE88
:10002000ADDEADDEADDEADDEADDEADDEADDE78
...
```

then using the `-break=16` option and an addressing value of 1 will change the output to:

```
:100000000EEF0AF03412ADDEADDEADDEADDEADDEFC
:06001000ADDEADDEADDE49
:10001600ADDEADDEADDEADDEADDEADDEADDE82
:10002600ADDEADDEADDEADDEADDEADDEADDE72
...
```

Here, the second record only specifies data up to address 15 and data in the following record begins at address 16.

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.

3.12 Ck Hexmate Option

The `-ck` option calculates a hash value over data in the HEX file and stores this value into the HEX file at a nominated address.

Attention: Note that if you are using MPLAB XC8 to build your project, it is often easier to use its `-mchecksum` option to control hash calculations performed by Hexmate.

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 inclusive hexadecimal address range over which the hash will be calculated. The interpretation of these addresses are based on the addressing value (see Addressing Hexmate Option). 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. The interpretation of this address is based on the addressing value (see Addressing Hexmate Option). 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. It is specified in bytes for most algorithms, but in bits for SHA algorithms. See the Hexmate Hash Algorithm Selection table below for allowable widths. 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. The width must be 0 or a value 2 or greater. Typical values would be 2 or 4, depending on the application. It is recommended that you do not exceed a width of 8. 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. In hash calculations, to avoid processing the 'phantom' 0x00 bytes added to HEX files for PIC24 and 24-bit instruction set dsPIC devices, use s4.
- *XORvalue* is a hexadecimal value that will be XORed with the hash result before it is stored.

Table 3-2. Hexmate Hash Algorithm Selection

Selector	Algorithm description	Allowable hash width
-5	Reflected cyclic redundancy check (CRC)	1 - 8 bytes
-4	Subtraction of 32 bit values from initial value	1 - 8 bytes
-3	Subtraction of 24 bit values from initial value	1 - 8 bytes
-2	Subtraction of 16 bit values from initial value	1 - 8 bytes
-1	Subtraction of 8 bit values from initial value	1 - 8 bytes
1	Addition of 8 bit values from initial value	1 - 8 bytes
2	Addition of 16 bit values from initial value	1 - 8 bytes
3	Addition of 24 bit values from initial value	1 - 8 bytes
4	Addition of 32 bit values from initial value	1 - 8 bytes
5	Cyclic redundancy check (CRC)	1 - 8 bytes
7	Fletcher's checksum (8 bit calculation)	2 bytes
8	Fletcher's checksum (16 bit calculation)	4 bytes
10	SHA-2	256 bits
11	SHA-1	160 bits

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
```

If the addressing value is set to1, this option will calculate a checksum (16-bit addition) over the HEX file address range 0 to 0x1FFF(inclusive) 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 use words that are aligned to the addresses in the HEX file, 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 through 5. If it is acting on the HEX record (data underlined):

```
:10000000 64002500030A75007600770078006400 1C
```

the hash will be calculated from the (hexadecimal) bytes 0x64, 0x25, and 0x03 (the two 0x00 bytes are skipped). Processing the same HEX record with an option that uses a different start and end address range (1 through 6):

```
-ck=1-6@100w2g5p1021s2
```

the hash will be calculated from the (hexadecimal) bytes 0x25, 0x03, and 0x75 (the same 0x00 bytes are skipped). These features attempt to mimic data read limitations of code running on the device, and thus the words they use are aligned with actual addresses.

If a non-zero *skipWidth* has been specified for algorithms that process 16 or 32 bits of data per iteration of the hash algorithm, then the skipped bytes are padded with 0 bytes. For example if the HEX record was (data underlined):

```
:04000000 12345678 E8
```

then employing a 16-bit additive checksum (algorithm 2) over this data would normally add 0x3412 and 0x7856. If a *skipWidth* of 2 was requested, the algorithm would add 0x0012 and 0x0056, since every second byte in the input was skipped. However, if the CRC (algorithm 5), which processes the input sequence one byte per iteration, had instead been selected, the bytes 0x12 and 0x56 will be processed by the CRC algorithm.

Note that this option inserts data into the output HEX file, thus, to prevent a conflict, you must ensure that the locations where this data is to be placed are not already populated by other data from a HEX file or from another option.

The following are Windows examples of some complete Hexmate commands that calculate a hash value and that can be executed as a post-build step, as specified in the **Building** category of the MPLAB X IDE **Project Properties** dialog. They assume that Hexmate is in the execution path. If that is not the case, add the path to the Hexmate application.

The command:

```
hexmate.exe ${ImagePath} -FILL=0xFF@0x0:0x17FEF -ADDRESSING=2 -CK=0-17FEF@17FF0w-2g5p1021
-OMergeddsPIC.hex -LOGFILE=finaldsPIC.hxl
```

will calculate a CRC hash over the dsPIC device program memory address range 0x0-0x17FFF and store the hash at address 0x17FF0. Unused memory is filled. Note the use of the `-ADDRESSING` option to allow device addresses to be specified in the options.

The command:

```
hexmate.exe ${ImagePath} -FILL=0xFF@0x9D000000:0x9D07FFEF -ADDRESSING=1
-CK=9D000000-9D07FFEF@9D07FFF0+FFFFw-2g5p1021 -OMergedPIC32.hex -LOGFILE=finalPIC32.hxl
```

will calculate a CRC hash over the PIC32 device program memory address range 0x9D000000-0x9D07FFEF and store the hash at address 0x9D07FFF0.

See Hash Value Calculations for more details about the algorithms that are used to calculate hashes.

Attention: The MPLAB X IDE calculates CRC32 hash values for HEX files produced when you build a project. These hashes are calculated from every byte in the HEX file and can verify the integrity of the entire file. As Hexmate calculates a runtime-checkable hash value from values stored only in HEX file Data records, hash values it calculates will never agree with those produced by the IDE. These hashes are intended for different purposes.

3.13 Fill Hexmate Option

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` will fill every unused byte between address 0 and 0xF with the byte value 0x55, for example:

```
:10000000FBEF3FF05555555555555555555555555DB
```

whereas `-fill=w2:0x55@0:0xF` will fill every unused byte between the same addresses with the value 0x0055, for example:

```
:10000000FBEF3FF05500550055005500550055005500D9
```

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, the bytes are stored in little-endian byte order. The following show the possible fill expressions:

- *const* fill memory with a repeating constant; for example, `-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; for example, `-fill=0xBEEF+=1@0:0x1FF` attempts to fill 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; for example,`-fill=0xBEEF-=0x10@0:0x1FF` attempts to fill 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; for example, `-fill=0xDEAD,0xBEEF@0:0x1FF` will fill with 0xDEAD, 0xBEEF, 0xDEAD, 0xBEEF, etc., for example:

  ```
  :10000000FBEF3FF0ADDEEFBEADDEEFBEADDEEFBE2F
  ```

@address fills a specific address with *fill_expr*. The interpretation of this address is based on the addressing value. 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 to be filled with *fill_expr*. The interpretation of this address is based on the addressing value. 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.

For Baseline and Mid-range PIC devices, as well as all 24-bit PIC and DSC devices, you can, if desired, specify a value of 2 with Hexmate's `-addressing` option so that the addresses used in this option will be device addresses. For all other devices, use the default addressing value or specify an addressing value of 1.

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@0x11001:0x1100c
```

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.

```
:0C100100221122112211221122112211B1
```

Compare that to when the option is `-fill=w1:0x11,0x22@0x11000:0x1100c`, which does specify a starting address that is a multiple of 2.

```
:0D100000112211221122112211221122 11A0
```

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.

3.14 Find Hexmate Option

The `-find=`*`opcode`* option is used to detect and log occurrences of an opcode or code sequence in the Data field of data records in the HEX file. 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 `F101`, for example:

  ```
  :100FE800F5007C310800F001F10120000030860096
  ```

  .
- *`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`*, whose interpretations are based on the addressing value (see Addressing Hexmate Option), are addresses that 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.

For example, 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. The `w` character requests that a warning be issued each time this sequence is found.

If the option was instead, `-find=1234M0F00@0-7FFF/2wt"ADDXY"`, 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 (see Logfile Hexmate Option, 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 in the following sections).

3.15 Find And Delete Hexmate Option

If the `delete` form of the `-find` option (see Find Hexmate 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
```

Attention: This function should be used with extreme caution and is not normally recommended for the removal of data or executable code.

3.16 Find and Replace Hexmate Option

If the `replace` form of the `-find` option (see Find Hexmate 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
```

Attention: This function should be used with extreme caution and is not normally recommended for the replacement of data or executable code.

3.17 Format Hexmate Option

The `-format=`*type*`[,`*length*`]` option specifies the format of the Intel HEX file to output and optionally the maximum record data length in the output file.

A simple use-case of this option is to change the format of a HEX file; for example, from an INHX16 to an INHX32 format. It can also be used to specify the output file type after merging input HEX files (possibly with different formats) and/or inserting or adjusting values into the data.

The *type* argument specifies a named INHX format to generate, such as `INHX16`, as tabulated below. These formats are discussed in HEX File Formats. Note that the address range of data to be output might dictate which formats can be used.

The *length* argument is optional and sets the maximum number of bytes per data record. A valid length for this option is between 1 and 255 decimal, inclusive, with 16 being the default if a length is not specified with the option.

The possible formats that are supported by this option are listed below. Note that the INHX032 selection is not an actual named INHX format. Specifying this format generates an INHX32 format file, but it will also initialize the upper base address to zero using an Extended Linear Address (type 4) record. This initialization is a requirement of some device programmers.

Table 3-3. HEX file Formats

Type	Valid record types	Comments
INHX8M	0, 1	16-bit wide HEX file address field
INHX16	0, 1, 2, 3	20-bit wide HEX file address field
INHX32	0, 1, 4, 5	32-bit wide HEX file address field
INHX032	0, 1, 4, 5	INHX32 with initialization of Extended Linear Address to zero

3.18 Help Hexmate Option

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.

3.19 Logfile Hexmate Option

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.

3.20 Mask Hexmate Option

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 (inclusive), whose interpretation is based on the addressing value (see Addressing Hexmate Option). All values are assumed to be hexadecimal. Multibyte mask values can be entered in little endian byte order.

3.21 O: Specify Output File Hexmate Option

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.

3.22 Serial Hexmate Option

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. Its interpretation is based on the addressing value (see Addressing Hexmate Option).
- *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.

Note that this option inserts data into the output HEX file, thus, to prevent a conflict, you must ensure that the locations where this data is to be placed are not already populated by other data from a HEX file or from another option.

For example:

```
-serial=000001@EFFE
```

will store HEX code 0x00001 to address 0xEFFE, assuming an addressing value of 1.

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.

3.23 Size Hexmate Option

Using the `-size` option will report to standard output the total number of bytes of record data within the resultant HEX image. This will be the total size of all bytes in the Data/Argument field of each record. The size will also be recorded in the log file if one has been requested. For example:

```
> hexmate -ooutput.hex float.hex -size
Total data: 988 (3DCh) bytes
> hexmate -ooutput.hex float.hex -string@9000="More data" -size
Total data: 998 (3E6h) bytes
```

3.24 String Hexmate Option

The `-string` option will embed into the HEX file 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. Its interpretation is based on the addressing value (see Addressing Hexmate Option).
- *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, for example.
- *Text* is the string to convert to ASCII and embed.

Note that this option inserts data into the output HEX file, thus, to prevent a conflict, you must ensure that the locations where this data is to be placed are not already populated by other data from a HEX file or from another option.

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, assuming an addressing value of 1.

And again:

```
-string@1000t34="My favorite string"
```

will store the same string, trailing every byte in the string with the HEX code 0x34, which would encapsulate the bytes into a `retlw` instruction if targeting a Mid-range PIC device, for example.

3.25 Strpack Hexmate Option

The `-strpack=`*`spec`* option performs the same function as `-string`, but with two important differences, described below.

Whereas the `-string` option stores the full byte corresponding to each character, the `-strpack` option 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.

Note that this option inserts data into the output HEX file, thus, to prevent a conflict, you must ensure that the locations where this data is to be placed are not already populated by other data from a HEX file or from another option.

3.26 W: Specify Warning Level Hexmate Option

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.

4. 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 implements several hash algorithms, such as checksums and cyclic redundancy checks, which can be selected to calculate a hash value of a program image that is contained in a HEX file. This value 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's `-ck` option requests that a hash be calculated, as described in Ck Hexmate Option.

Some consideration is required when a hash value is being calculated over memory that contains unused memory locations. Consider using Hexmate's `-fill` option (see Fill Hexmate Option) to have these locations programmed with a known value. Avoid filling the locations where the hash value will be stored, as memory is filled before the hash is calculated and this can result in an error.

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. Use the `s2` suboption to Hexmate's `-ck` option to have the MSB of each 2-byte word skipped. Note, however, that this sort of verification process will not detect corruption in the MSB of each program word.

To accommodate the 24-bit (3 byte) program memory word size on 24-bit instruction set dsPIC and PIC24 devices, the compiler inserts a 0x00 phantom byte after each 3-byte instruction to make up a 4-byte word. Hexmate will normally see and process these phantom bytes when calculating a hash value, whereas code running on the device to perform the same calculation might not. When executing Hexmate explicitly, use the `s4` suboption to the `-ck` option to have the MSB of each 4-byte word skipped for these devices.

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. Use the `r2` suboption to Hexmate's `-ck` option to have Hexmate process the 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. Use the `t34` suboption to Hexmate's `-ck` option to have Hexmate store each byte of the hash value in a

`retlw` instruction. The `retlw` instruction is encoded as `0x34`*`nn`*, where *`nn`* is the 8-bit data value to be loaded to WREG when executed.

If you are targeting a 24-bit PIC device, where the 24-bits of program memory associated with an instruction appear as 3 bytes of data and 1 phantom byte in the HEX file and, for example, you wanted to have a 32-bit hash stored in only the least significant 16-bit word of each location, use Hemxate's `t0000.2` suboption to the `-ck` option to 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.

4.1 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.

4.1.1 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 from 1 to 8 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.

4.1.2 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 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
```

4.1.3 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 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;
}
```

4.1.4 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.

4.1.5 SHA Algorithms

Hexmate implements secure hash algorithms (SHA). The selector values 10 and 11, select the SHA-2 and SHA-1 algorithms respectively.

The code to implement a SHA is more complex than other algorithms supported by Hexmate. Public-domain implementations of these algorithms are available for download from third-party websites, such as github.com/B-Con/crypto-algorithms.

Hexmate can generate a 160-bit (20 byte) wide SHA-1 hash, or a 256-bit (32 byte) wide version of the SHA-2 hash, referred to as SHA-256.

Here is how Hexmate might be used when, for example, a SHA256 hash value is to be calculated over the address range 0x0 to 0x1FF. The result will be stored at a starting address of 0x1000 in little endian format.

```
-ck=0-1ff@1000g10w-256
```

5. Error and Warning Messages

Listed here are error, warning, and advisory messages produced by Hexmate, with an explanation of each message.

Messages are assigned a unique number, which appears in brackets before each message description. The number is also printed by Hexmate when the message is issued. The messages shown here are sorted by their number. Only numbered messages issued by Hexmate are listed. A * in the message will be replaced by Hexmate with a string that is specific to that particular message.

5.1 Messages

(294) can't allocate * bytes of memory
This is an internal application error. Contact Microchip Technical Support with details.

(378) can't create * file "*"
This type of file could not be created. Is the file, or a file by this name, already in use?

(636) error in Intel HEX file "*" on line *
An error was found at the specified line in the specified Intel HEX file. The HEX file may be corrupt.

(941) bad "*" assignment; USAGE: **
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 "*"
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 *
Sources to Hexmate requested differing data to be stored to the same address. To force one data source to override the other, use the '+' specifier with that source. 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
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
The requested byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the -CK option.

(949) start of range must be less than end of range
The -CK 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
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: *
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
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
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 "*"
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 *
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
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
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
This option currently is not available and will be ignored.

(966) no END record for HEX file "*"
Intel HEX file did not contain a record of type `END`. The HEX file can be incomplete.

(1030) Hexmate - Intel HEX editing utility (Build 1.%i)
Indicating the version number of the Hexmate being executed.

(1031) USAGE: * [input1.HEX] [input2.HEX]... [inputN.HEX] [options]
The suggested usage of Hexmate.

(1032) use –HELP=<option> for usage of these command line options
More detailed information is available for a specific option by passing that option to the `-HELP` option.

(1033) available command-line options:
This is a simple heading that appears before the list of available options for this application.

(1034) type "*" for available options
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.

(1198) too many "*" specifications; * maximum
This option has been specified too many times. If possible, try performing these operations over several command lines.

(1200) Found %0*lXh at address *h
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

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: *

An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated

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

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.

(1212) Found * (%0*lXh) at address *h

The code sequence specified in a `-FIND` option has been found at this address.

(1245) value greater than zero required for *

The align operand to the Hexmate `-FIND` option must be positive.

(1600) "*" argument : *

There is an error in an argument to a Hexmate option. The message indicates the offending argument and the problem.

(1601) "*" argument : *

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 (*)

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.

(1606) w[no-]error option has been passed an invalid or missing message identifier (Hexmate)

The --werror option is missing a message number after the = character or has been passed something that is not a number.

```
hexmate --werror=off -format=inhx32 main.hex -ofinal.hex
```

(2059) conflicting * register values found in Start Segment Address record (3)

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

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

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 Hexmate driver, this is specified using the `revword` suboption to `-mchecksum`.

(2062) word width must be * when specifying reserve byte order hash

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 Hexmate 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

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 Hexmate's `-mchecksum` option.

(2071) could not find record containing hash starting address 0x*

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)

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.

(2074) word size for byte skip with hash calculation must be *

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

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 *

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

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 (*)

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 (*)

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.

6. Document Revision History

Revision A (September 2020)

- Initial release of this document, adapted from content in the MPLAB® XC8 C Compiler User's Guide, DS 50002053.

Revision B (December 2022)

- Added description for `--ssa` option, used with INHX32 files.
- Indicated the new filename format used by message description files.
- Added new section indicating possible causes of failure for common actions.
- Updated the operation of the `--sla` option, used with INHX16 files.
- Clarified the operation of the `-ck` option and described the new XOR suboption.
- Improved the description of the `-fill` option.
- Improved description of the `-format` option.

Revision C (October 2023)

- Added description of the Intel HEX file specification, as interpreted by Hexmate.
- Discussed Intel HEX file formats used by Microchip tools.
- Updated and better presented the allowable hash widths for all supported algorithms.
- Added SHA-1 to list of support hash algorithms.
- Added new `--werror` option, which promotes warnings to errors.
- Added list of error and warning messages pertinent to Hexmate.
- Expanded Potential Causes of Failure section to include situations where software breakpoints have been used.
- Corrected maximum record length allowable with the `-format` option.
- General improvements to the descriptions of Hexmate operations and options.

Revision D (October 2024)

- Updated the permitted reverse word widths when calculating a hash using Hexmate.
- General typographical corrections and improvements.

Microchip Information

Trademarks

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

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

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, Clockstudio, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, EyeOpen, GridTime, IdealBridge, IGaT, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, IntelliMOS, Inter-Chip Connectivity, JitterBlocker, Knob-on-Display, MarginLink, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, mSiC, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, Power MOS IV, Power MOS 7, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SmartHLS, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, Trusted Time, TSHARC, Turing, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

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

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP "AS IS". MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED

WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Microchip Devices Code Protection Feature

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

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip product is strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.