AN204834

F²MC-16FX Family, CRC16/Checksum Calculation for Flash

This application note describes a method to calculate a CRC/Checksum code both during development and at run-time and compare them.

Contents

1 Introduction

The content of the Flash memory can be checked for accidental alterations by adding a CRC code or a simple checksum code. At run-time the CRC/Checksum code can be calculated again and compared against the previously stored one. This allows detection of accidental alterations.

This application note describes a method to calculate a CRC/Checksum code both during development and at run-time and compare them.

There is an archive available that contains all software discussed in this application note. It contains both a project for Softune Workbench and the utility *CRC_calculator.exe* for MS Windows. Please see chapter 4 for download link.

2 Motivation of CRC and Checksum Usage

CRC/Checksum codes can be used to check Flash memory for unintended alterations.

The reliability of an application depends amongst others on the correctness of the program memory. Any unintended alteration of the program memory will inevitably lead to a malfunction of an application. Such an unintended alteration may happen for example by a drop of supply voltage during programming or exposure to high temperatures of the program memory. The program memory of a microcontroller is typically a built-in Flash memory. Several standards for safety critical applications recommend using CRC checks for the program memory. Sometimes a (quicker) simple Checksum test is sufficient.

One method to achieve a high confidence level of having the intended program data available is usage of a cyclic redundancy check (CRC). In this method a CRC result is calculated over certain Flash memory areas. This result is then also stored in the same Flash memory. At run-time the CRC result is calculated again and compared against the pre-calculated result. If both results match, it can be assumed that the Flash memory content was not altered by accident. The same is done for simple Checksum testing.

The CRC result is usually calculated by a polynomial division. Several different polynomials are in use and scientific research has shown that some polynomials are better suited than others. Some of the better suiting polynomials was appraised by the CCITT and are known as the CRC-CCITT. There are different CRC-CCITT available that differ in their size and consequently in the error burst lengths that can be detected. In this application note, the CRC-CCITT (CRC16) is used. The corresponding polynomial is $x^{16}+x^{12}+x^5+1$, which represents the value `0x1021` in binary notation (x^{16} ignored for 16 bit).

The start value of the CRC/Checksum calculation may be chosen arbitrarily. If it starts with zeros, it may happen that errors in leading zeros in the data will not be discovered. In this application note and corresponding software, the start value is set to `0xFFFF`, which allows detection of alterations even for leading zeros. For simple Checksum calculation the start value does not have such consequences.

The address location of the CRC/Cheksum result can be inside of the memory range that the CRC code is calculated for. During calculation its address is skipped.

3 Implementation

Implementation of the Check Algorithms and Calculator Tool Usage

3.1 Concept

Checking the content of the Flash memory by a CRC16/Checksum code consists of several steps:

- The application contains an algorithm to calculate a CRC16/Checksum result and compare it with a previously calculated result.
- An application to calculate the CRC16/Checksum during development.
- The CRC16/Checksum calculated during development must be stored at a well-defined location in Flash memory.
- During debugging, the previously calculated CRC16/Checksum result must be loaded into the debug system.

The above mentioned items will be discussed in more detail in the following sections.

The CRC16/Checksum result is calculated usually over a certain Flash memory address range or ranges, no matter if this area is actually used by the application. This allows easy modification of the application without always updating the address range(s) manually. It must be ensured that there is a well-defined value used for the memory locations where no actual code is stored. This can be achieved by using the adjuster from Softune Workbench. See 3.3.6 for more details.

3.2 MCU code

An application can check the program memory for accidental alterations by cyclically performing a CRC16 code calculation or simple Checksum code of a desired address range. The calculated CRC16 result can be compared to a pre-calculated result and the matching result returned. A corresponding function may take pointers to the start and end addresses as well as to the pre-calculated code as input parameters. It returns 0 if both results match, i.e. it is assumed that the memory content is correct, and 1 if there is a mismatch detected.

```
unsigned char CalcCRC16(void)
{
   // Define Flash Blocks
   unsigned long flashblocks[] = FLASH_BLOCKS;
   unsigned char blocks;                    // Flash block counter
   unsigned int  checked_data = CRC16START; // Start value of CRC16/Check-Sum
   unsigned long start_address;             // start address of current block
   unsigned long end_address;               // end address of current block
   unsigned long address;                   // current address to be calculated
   unsigned char data;                      // current data to be calculated

   // Go through all user Flash blocks to be calculated
   for (blocks = 0; blocks < (sizeof(flashblocks) / (2 * sizeof(long)));
        blocks++)
   {
      // Get start and end addresses of current Flash block
      start_address = flashblocks[2 * blocks];
      end_address   = flashblocks[2 * blocks + 1] + 1;

      for (address = start_address; address < end_address; address++)
      {
         // skip CRC16/Checksum itself
         if ((address != (unsigned long)ADDR_CRC16_CHECKSUM) &&
             (address != (unsigned long)ADDR_CRC16_CHECKSUM + 1))
```

```
         {
            data  = *(__far unsigned char*)address;

            #if (CHECK_ALGORITHM == CHECK_CRC16)
               // Calculate CRC16 for each char data in current Flash block
               checked_data = crc16calc(checked_data, data);

            #elif (CHECK_ALGORITHM == CHECK_SUM)
               // Calculate Checksum for each char data in current Flash block
               checked_data = checksumcalc(checked_data, data);

            #endif
         }
      } // for address
   } // for blocks

   // return 0: OK, 1: Error
   return (checked_data != *(__far unsigned int*)ADDR_CRC16_CHECKSUM);
}
```

The pink background code is used, when a CRC16 calculation is selected, where green shows the Checksum calculation.

The following code box shows the algorithms itself.

```
#if (CHECK_ALGORITHM == CHECK_CRC16)
      unsigned int crc16calc(unsigned int crc16, unsigned char data)
      {
         signed char bit;                          // Bit counter

         // Go through all bits of 'data'
         for (bit = 8; bit > 0; bit--)
         {
            // Higher bit of current 'crc16' value does not match to
            //   current bit of 'data'?
            if (((crc16 & 0x8000) ? 1 : 0)
                 != ((data & (1 << (bit - 1))) >> (bit - 1)))
            {
               // Shift-left 'crc16', XOR with Polynomial
               crc16 <<= 1;
               crc16 ^= POLYNOMIAL;
            }
            else
            {
                // Shift-left 'crc16' only
                crc16 <<= 1;
            }
         }

         return crc16;
      }

      #elif (CHECK_ALGORITHM == CHECK_SUM)
      unsigned int checksumcalc(unsigned int checksum, unsigned char
      data)
      {
          checksum += data;       // overflow of adding is ignored

          return checksum;
      }

      #endif
```

The following definition has to be set also in a appropriate header file:

```
// ---------------------------------------------------------------------
// System definitions

// section preprocessor argument workaround for address
//
#define NESTED_MACRO(name) #name
#define SECTION_LOCATE(addr) NESTED_MACRO(addr)

// predefined macro values for used algorithm
//
#define CHECK_CRC16 0
#define CHECK_SUM   1

// ------------------------- User Edit --------------------------------

// Define algorithm for Flash checking
//
#define CHECK_ALGORITHM CHECK_SUM

// Flash blocks start and end addresses
//
// Format {<start-address[0]>, <end-adress[0]>, <start-address[1]>, ...,
//         <end-address[n-1]>, <start-address[n]>, <end-address[n]>}
//
// Note, that the CRC16/CS calculation is done in the order of the blocks
// defined here. The addresses must be ascending to get the same results
// as from the DOS tool 'CRC_calculator.exe'. Up to 10 blocks are allowed.
//
#define FLASH_BLOCKS {0xF80000, 0xF8007F, \
                      0xF80100, 0xF801FF}

// Flash address of the CRC16/Checksum result (may also be included in Flash
// blocks!)
//
#define ADDR_CRC16_CHECKSUM 0xFF0008

// CRC16 Polynomial
//
#define POLYNOMIAL 0x1021

// CRC16/Checksum Start Value
//
#define CRC16START 0xFFFF

//
// ------------------------End of User Edit -----------------------------
```

The pre-calculated CRC16/Checksum code must be stored at a well-defined location in Flash memory. The CRC16/Checksum used in this application note generates a 16-bit value.

The CRC16/Checksum result value is stored here by an assembly instruction, because the usual C `#pragma section` directives do not allow to parameterize the address in the locate attribute. With the workaround of the nested macro above, the `.SECTION` pseudo code allows this parameterizing for its locate attribute:

```
__asm("  .SECTION CRC16, CONST, LOCATE=" SECTION_LOCATE(ADDR_CRC16_CHECKSUM) );
__asm("  .DATA.W  0xFFFF");
```

The CRC16/Checksum can either be placed instead the `0xFFFF` or the debug memory will be patched by the result value by procedure file.

The CRC16/Checksum code can be calculated at runtime by calling function CalcCRC16().

```
void main(void)
{
    ...

    if (CalcCRC16())
    {
        // Error handling here ...
    }
    else
    {
        // Everything fine here ...
    }

    ...
}
```

3.3 CRC16/Checksum Value Calculation on Windows

The CRC16/Checksum result value must be calculated during development and stored in the application. A command line program is available that can calculate the required CRC16/Checksum value by parsing the Motorola S-format file that contains the application. The CRC16/Checksum value is either output on the command line or can be placed automatically at the desired location in the Motorola S-format file. In addition, the program can generate a procedure file that can be used to load the CRC16/Checksum value in the Softune Workbench simulator or the emulation system.

The CRC16/Checksum value calculation program uses the CRC-CCITT (CRC16) polynomial $x^{16}+x^{12}+x^5+1$ for CRC16, a simple adding algorithm for simple Checksum and a start value of `0xFFFF`.

3.3.1 Usage

The command line program has following syntax when using the parameters by command line itself:

```
CRC_calculator \
    -ran <Flash Block 1 Start Address>,<Flash Block 1 End Address>\
         [,<Flash Block 2 Start Address>,<Flash Block 2 End Address>\
         [,<..>,<..>]]\
    -mhx [Path]<MHX File Name>\
   [-p <CRC16/Checksum Address>]\
   [-d   [Path]<Procedure File Name>]\
   [-a   <Check Algorithm: (default:CRC16/CHECKSUM)>]
```

Parameter	Description	Examples

`-ran <start address 1>,\ <end address 1>\ [,<start address 2>,\ <end address 2>,<…>,<…>]`	Start address *n*: Address of a Flash block to be checked from that on the CRC16/Checksum will be calculated
	End address *n*: Last address of a Flash block that the CRC16/Checksum will be calculated for
	Must be larger than Start address and following block addresses must be ascending.
	Up to 10 blocks can be specified.
		`-ran 0xDF0000,0xDF5FFF`
		
		`-ran 0xDF7000,0xDF7FFF, 0xF80000,0xF9FFFF, 0xFF8000,0xFFFFFF`

`-mhx <MHX file name>`	The name and path (optional) for the MHX file to be calculated and patched by `-p` option.	`-mhx MyFile.mhx`
		
		`-mhx Z:\project\ABS\ Test.mhx`

`[-p <location of CRC>]`	Do not output CRC value to command line but insert it into Motorola S-format file at specified address	`-p 0xDF7FFE`
		
		`-p 0xF81111`

`[-d <procedure file name>]`	In addition to inserting the CRC value into Motorola S-format file, generate a procedure file that loads the CRC value in simulator or emulator	`-d crc16_patch.prc`

`[-a <check algorithm>]`	Use `CRC16` for CRC16 calculation or `CHECKSUM` for Checksum calculation
	Default is CRC16 when not specified
		`-a CRC16`
		
		`-a CHECKSUM`

The tool also can be called by an option file. This option file has to be specified as a single argument. The syntax of the parameters is the same as for the command line.

Example:

```
CRC_calculator options.txt
```

The *option.txt* file could contain then the following parameters:

```
-ran 0xFF0002,0xFF000D -mhx abs\project.mhx -a CHECKSUM -p 0xFF0008 -b prc\crc16_patch.prc
```

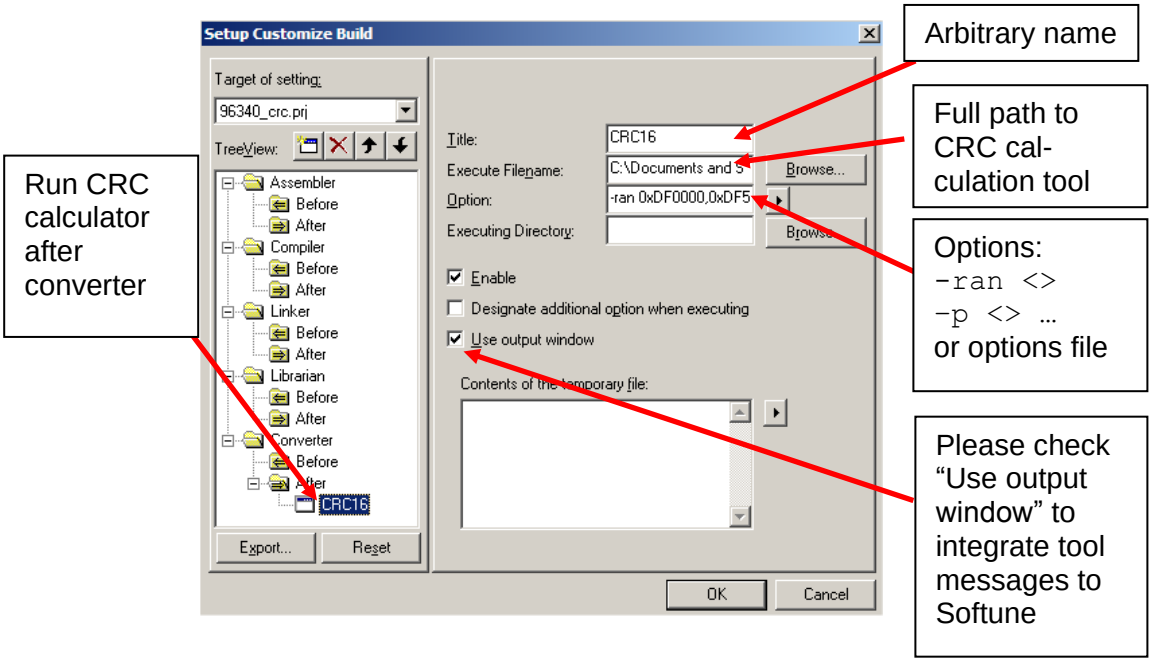
Note: A <RETURN> character at the end of the line is not necessary.

3.3.2 Errors

The CRC16/Checksum tool may throw several error messages in case of wrong parameters and parameter specifications. These errors and their root causes are listed in Appendix A (→4).

3.3.3 Integration in Softune Workbench tool chain

The CRC16/Checksum value generation tool can be run automatically by Softune Workbench whenever a new software release is build. Open the dialog *Customize Build…* from the *Project* menu. This opens a new window, which shows on its left hand side the individual tools of the build process. The CRC16/Checksum calculation tool should be run after the converter. All paths should be specified relative to the tool location (e.g. root directory of project).



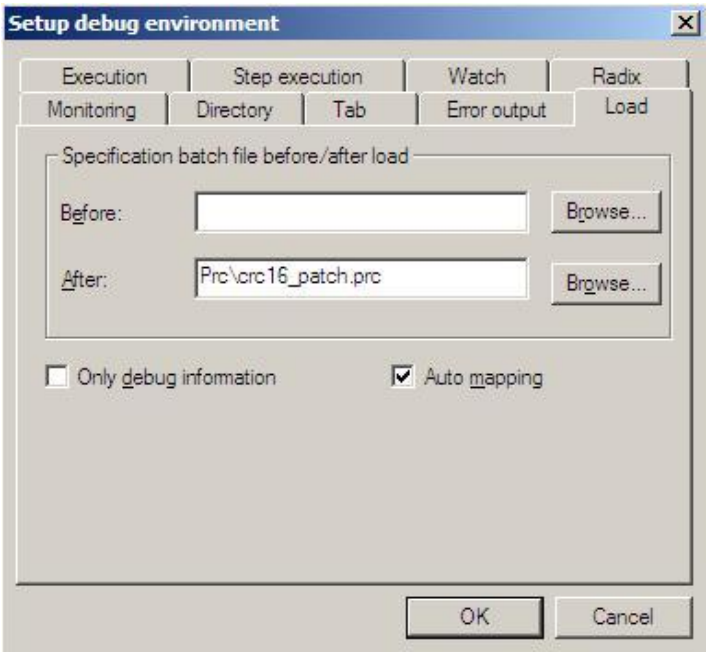
3.3.4 Softune Workbench debugger

The Softune Workbench simulator and emulator debugger both load the absolute linker output file. The CRC16/Checksum value calculation program however can only place the CRC16/Checksum value in the Motorola S-format file. Hence, the CRC16/Checksum value must be loaded separately into a debug session. The calculator tool offers the option to generate a procedure file that can be loaded in Softune Workbench debuggers. The procedure file contains following content

```
set memory /word <CRC address> = <CRC value>
```

where `<CRC address>` is the address given by the `-p` parameter and `<CRC value>` is the calculated value.

Procedure files can be loaded manually by opening the Command Window and issuing the command `batch <procedure file>`. It can also be loaded automatically whenever the target file is loaded. This can be achieved in the *Setup → Debug environment → Debug environment* on tab *Load*. The procedure file should be loaded after loading the target file:



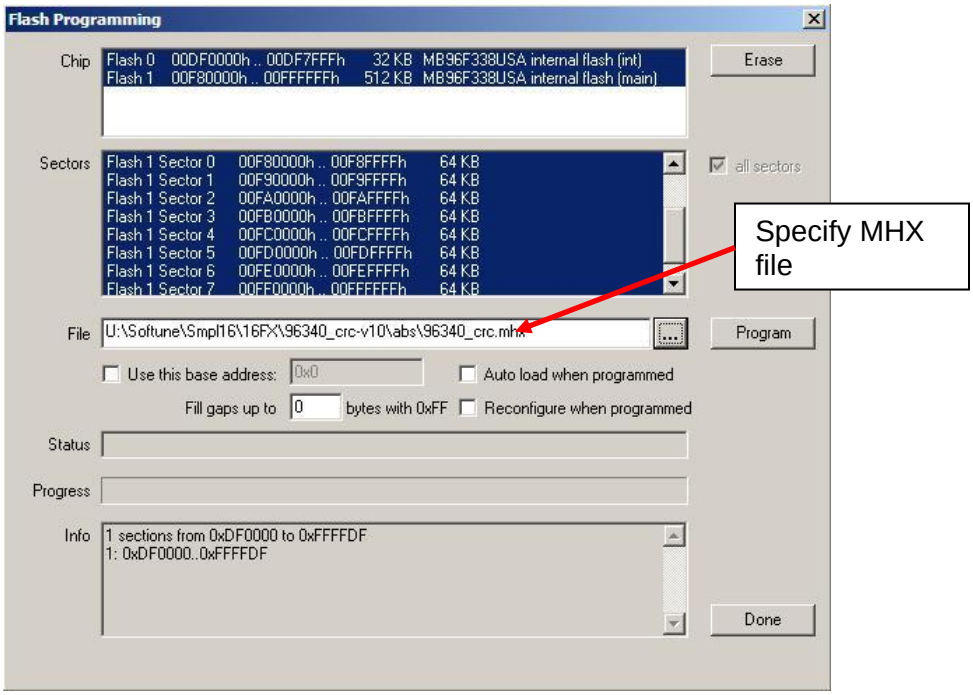
If more than one procedure file should be executed, e.g. in addition the procedure file to simulate the ROM mirror in the simulator, an additional procedure file must be used. This procedure file is executed by Softune Workbench and it runs the other procedure files. Such an intermediate procedure file could look like this:

```
# file name: load_procedure_files.prc
# This loads several other procedure files

batch prc\romconst.prc
batch prc\crc16_patch.prc
```

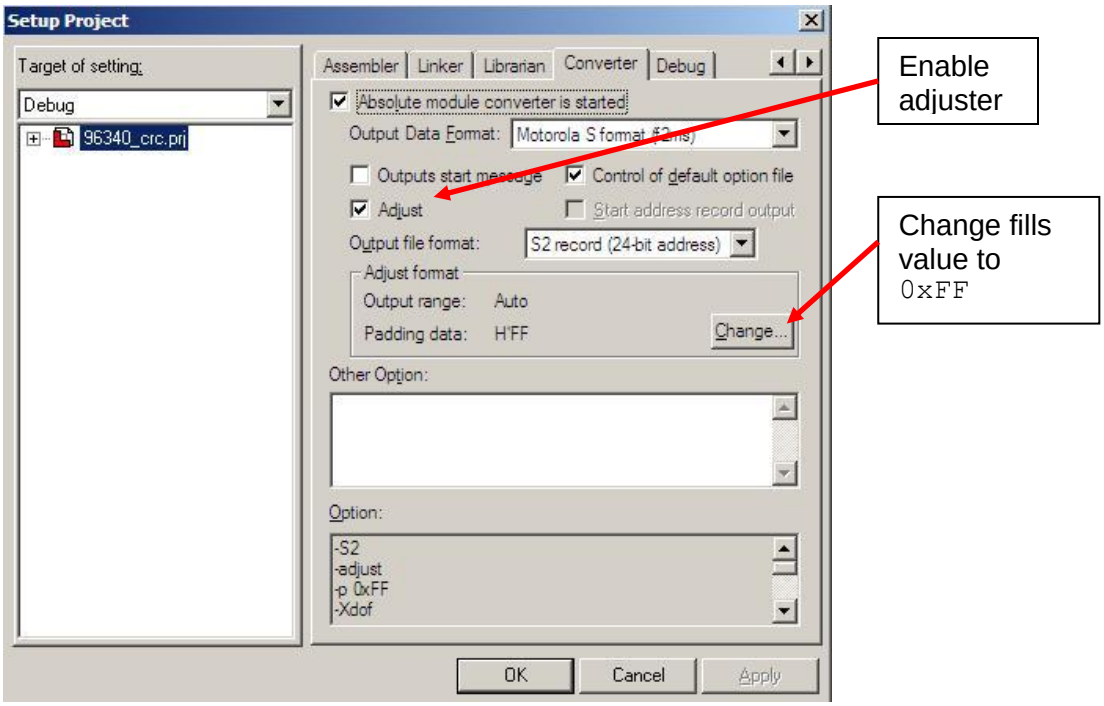
3.3.5 EUROScope debugger

The EUROScope debugger offers different ways to load a new software compilation and program it into the Flash memory of the MCU. The most common method is to use *File → Open Application and Download* or respective button on the toolbar. This loads the absolute symbol file and programs the data stored in this file into the Flash memory. This method unfortunately does not allow inserting the CRC16/Checksum value. If the CRC16/Checksum value should be inserted, please load the new software into EUROScope using *File → Open Application*. Programming the software into Flash memory can be done using the option *Tools → Flash*…. This opens a dialog window that allows specifying the Motorola S-format file.



3.3.6 Softune adjuster

The adjuster of Softune Workbench can be used to fill undefined memory areas or simply every ROM memory gap. It can be configured in the dialog *Project* → *Setup project* → tab *Converter*. First, it must be enabled. Secondly, one can choose a special fill value and address range. By default, all unused Flash memory is filled by `0xFF`.



It is strongly recommended to use this adjuster, because the CRC16/Checksum calculator tool does not check, if every address is actually contained in the MHX file. It assumes that the user took care of this requirement. The adjuster grants this.

4 Appendix A – Error Messages of Calculator

Error Messages

4.1 List of all Error Messages of the CRC16/Checksum Calculator Tool

Error #	Error Text	Explanation
100	`Procedure file name missing`	The '`-p`' option was specified without a following file name.
101	`MHX file name missing`	The '`-mhx`' option was specified without a following file name.
102	`MHX file not found: <name>`	The specified MHX file was not found
103	`MHX file error`	An error occurred parsing the MHX file
104	`MHX file does not contain all Flash block data`	The list of Flash blocks specified by '`-ran`' option was not processed completely after MHX file scanning
105	`Cannot create procedure file`	A system error occurred while writing the procedure file specified by '`-p`' option
106	`Options file error`	An error occurred trying to parse the options file
200	`Unknown option: <option>`	An unknown option was specified
201	`Address range error`	An address larger than `0xFFFFFF` was specified
202	`Wrong address separator in '-ran' option`	A different separator than '`,`' was used in '`-ran`' option's Flash blocks
203	`Too many Flash blocks specified`	There were more than 10 Flash blocks specified in '`-ran`' option
204	`To few Flash address blocks`	The number of specified addresses are odd (not a block address pair specified)
205	`Addresses range error`	Violation in ascending addresses in '`-ran`' option
206	`Unknown check algorithm option: <option>`	Neither `CRC16` nor `CHECKSUM` was specified for '`-a`' option
207	`Too few options in options file`	The options file contains less than 4 mandatory options (`-ran <list>, -mhx <file>`)
300	`CRC16/Checksum address not contained in MHX file`	The tool could not find the address specified by '`-p`' option in the MHX file

5 Additional Information

Information about Cypress Semiconductor can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

The software example related to this application note is:

Mcu-an-300253-e-vXX-16fx_CRC_calculation_for_Flash_memory.zip

It can be found on the following Internet page:

http://www.cypress.com/16lx

Document History

Document Title: AN204834 - F²MC-16FX Family, CRC16/Checksum Calculation for Flash

Document Number: 002-04834

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	03/18/2010	Initial release
			04/20/2010	Updated(SW version)
			08/11/2010	Completely revised version
*A	5084250	NOFL	04/14/2016	Migrated Spansion Application Note MCU-AN-300253-E-V20 to Cypress format
*B	5865580	AESATP12	08/30/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709