

# Using the CRC Module on Hercules™-Based Microcontrollers

Christian Herget

## ABSTRACT

This application report describes how to use the embedded CRC module found on all Hercules devices, as well as how to calculate a signature for non-volatile memory with the TI ARM® Code Gen Tools Linker.

Project collateral and source code discussed in this application report can be downloaded from the following URL: <http://www.ti.com/lit/zip/spna235>.

---

**NOTE:** TI assumes no liability that the discussed implementation and provided code are free from faults, compliant to certain coding guidelines, nor was it developed in accordance with certain standards. The implementer has to ensure and verify that the code conforms to appropriate rules and standards, and he has the sole responsibility to ensure and verify correct functionality in his application.

---

## Contents

1	Cyclic Redundancy Check (CRC) Controller Module .....	2
2	Linker Generated CRC Tables .....	4
3	Implementation .....	5
4	Results and Conclusion .....	10

## List of Figures

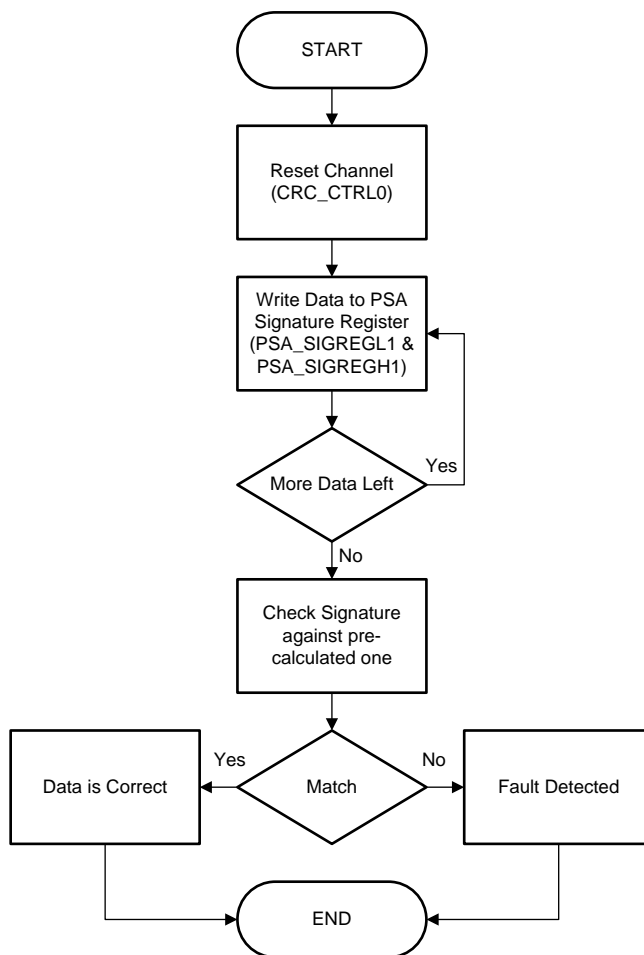
1	LFSR Diagram .....	2
2	Full-CPU Mode Flowchart .....	3
3	Full-CPU Mode With HALCoGen Flowchart .....	7
4	Semi-CPU Mode Polling Flowchart .....	8
5	Semi-CPU Mode Polling DMA Setup .....	9
6	Semi-CPI Mode Interrupt Driven Setup .....	9

## List of Tables

1	Examples Available per Device .....	6
2	Results Captured on RM48 (Example 1, 37kB) .....	10

Code Composer Studio is a trademark of Texas Instruments.  
ARM is a registered trademark of ARM Limited.  
All other trademarks are the property of their respective owners.





**Figure 2. Full-CPU Mode Flowchart**

The example for this mode is described in [Section 3.4](#).

### 1.2.3 Semi-CPU Mode

In semi-CPU mode, the DMA controller is utilized to perform data patterns transfer to the PSA Signature Register. The CRC controller generates a compression complete interrupt to CPU after each sector is compressed. Upon responding to the interrupt the CPU performs the signature verification by reading the calculated signature stored at the PSA Sector Signature Register and compares it to a pre-determined CRC value. There are two different examples using the Semi-CPU Mode, one uses a polling loop ([Section 3.5](#)) and the second one using interrupts ([Section 3.6](#)) to determine when the data compression has been finished.

### 1.2.4 Auto Mode

In AUTO mode, together the CRC Controller and the DMA controller can perform the data compression without CPU intervention. A sustained transfer of data to both the PSA Signature Register and CRC Value Register are performed in the background of CPU. When a mismatch is detected, an interrupt is generated to CPU. A 16-bit current sector ID register is provided to identify which sector causes a CRC failure.

In Auto Mode, the CRC module expects the memory to be divided in N equal sized sectors, where a pre-calculated CRC signature is associated to each of the N sectors. Two DMA channels are required to work in this mode. One is setup to copy the data values of N sectors into the CRC modules PSA Signature Register and could be triggered by the RTI time as for the Semi-CPU mode. The second DMA channel gets triggered by the DMA controller after a sector has been compressed and is used to copy the next CRC signature into the CRC Value Register (CRC\_REGx). As the pre-calculated CRC value gets automatically compared to the compressed data, the CRC module can now independently perform data verification of the entire (for example, program) memory.

However, the CRC Table implementation in the TI Linker (see [Section 2](#)) does not support this use case. Therefore, this is not further described in this document. However, the Semi-CPU Mode method suites most applications.

## 2 Linker Generated CRC Tables

The TI ARM Code Generation Tools Linker can be used to pre-calculate the signatures for different memory regions. This feature was introduced in 2011 with version 4.9.x of the TI ARM CGT tools. The CRC tables get configured in the Linker Command File (LCF) as part of the configured sections.

### 2.1 The `crc_table()` Operator

The operator `crc_table()` should be applied to any section in the LCF, which should be verified with a CRC. The Linker support different CRC algorithms can be specified with the `crc_table()` operator. However, the CRC module found in Hercules devices only support the so called TMS570\_CRC64\_ISO algorithm. The syntax for using the CRC tables in the LCF:

```
crc_table(user_specified_table_name[, algorithm=xxx])
```

The algorithm defaults to TMS570\_CRC64\_ISO. However, it might be a good idea to explicitly specify this to ensure code compatibility:

```
crc_table(user_specified_table_name, algorithm=TMS570_CRC64_ISO)
```

The following line shows an example on how to apply the `crc_table()` operator to a section in the LCF:

```
.const : {} palign=8, fill=0xffffffff, crc_table(_my_crc_table, algorithm=TMS570_CRC64_ISO)
```

In this case, the `crc_table()` operator is applied to the section `.const` that is used to hold global and static const variables that are explicitly initialized. A CRC table with the name `_my_crc_table` will be created. The same CRC table name can be used for many sections; the resulting CRC table will then hold several entries that can be accessed individually. The additional palign operator ensures that the section starts and ends on a 64-bit boundary. This is important keeping in mind, that the CRC module performs data compression on 64 bits. Furthermore the DMA controller can only read and write to addresses aligned to the read or write size. Thus, the sections should start and end on 64-bit boundaries.

An additional section should be added to the LCF to avoid warnings during the link step:

```
.TI.crctab : {} palign=8
```

The `.TI.crctab` section will hold all CRC tables created within the LCF. Omitting the lines above will cause the Linker to warn that an unknown section will be created. The section should be linked to a constant memory as it should not be modified.

### 2.2 Accessing the CRC Tables Within C Code

The Code Generation Tools include a header file to help to access the CRC tables within C code as part of the standard library. Include the `crc_tbl.h` file in every C source file, which accesses the CRC tables (`#include "crc_tbl.h"`). This header file holds the description (prototype) of the CRC tables. A CRC table can hold several entries called CRC records.

```

/*****
/* CRC Record Data Structure
/* NOTE: The list of fields and the size of each field
/* varies by target and memory model.
*****/
typedef struct crc_record
{
    uint64_t      crc_value;

```

```

uint32_t      crc_alg_ID;    /* CRC algorithm ID */
uint32_t      addr;         /* Starting address */
uint32_t      size;         /* size of data in bytes */
uint32_t      padding;      /* explicit padding so layout is the same */
                                   /* for COFF and ELF */
} CRC_RECORD;

/*****
/* CRC Table Data Structure */
*****/
typedef struct crc_table
{
uint32_t      rec_size;
uint32_t      num_recs;
CRC_RECORD    recs[1];
} CRC_TABLE;

```

There are several ways to define and access the CRC tables created by the linker. The following definition for a CRC table within the C code is recommended:

```
extern const CRC_TABLE _my_crc_table;
```

Note that the `_my_crc_table` name was also used in the example in [Section 2.1](#). The following code can be used to access the CRC table and the individual entries:

```

for (i = 0ul ; i < _my_crc_table.num_recs ; i++)
{
    /* Check for the right algorithm */
    if (TMS570_CRC64_ISO == _my_crc_table.recs[i].crc_alg_ID)
    {
        _my_crc_table.recs[i].crc_value;

        /* Convert Address to 64-bit Pointer */
        (uint64*)_my_crc_table.recs[i].addr;

        /* Adjust the size to be in 64-bit increment rather than bytes */
        _my_crc_table.recs[i].size / 8ul;
    }
    else
    {
        /* Unknown Algorithm */
    }
}

```

## 3 Implementation

### 3.1 Example Code

This application report comes with two Code Composer Studio™ projects to show the functionality of the CRC module. The supplied examples cover most of the devices out the Hercules Family. To see which of the two examples are available for which devices, see [Table 1](#). The first example shows the use of all three modes (SW, Full-CPU and Semi-CPU Mode); however, the Semi-CPU mode uses polling in this example and is not meant to be used that way in a production environment. The second example shows the Semi-CPU Mode in an interrupt driven fashion as it is intended to be used.

- Example 1: CRC SW Full-CPU and Semi-CPU Modes
  - Software Mode (see [Section 3.3](#))
  - Full-CPU Mode ([Section 3.4](#))
  - Semi-CPU Mode Polling ([Section 3.5](#))
- Example 2: CRC Semi-CPU Mode IRQ
  - Semi-CPU Mode Interrupt Driven ([Section 3.6](#))

**Table 1. Examples Available per Device**

	LS04	LS07	LS12	LS31	LC43	RM42	RM44	RM46	RM48	RM57
Example 1	x		x	x	x	x		x	x	x
Example 2			x	x	x			x	x	x

### 3.2 Linker Generated CRC Table

Linker generated CRC tables, described in [Section 2.1](#), are used in the example projects. For this, a Linker Command File (linker\_command\_file.cmd) was created. This Linker Command File also shows how to use the Linker Generated ECC feature of the TI Linker ([http://processors.wiki.ti.com/index.php/Linker\\_Generated\\_ECC](http://processors.wiki.ti.com/index.php/Linker_Generated_ECC)).

The generated CRC tables can also be seen in the so called Map file generated by the linker, for example:

```
_my_crc_table @ 00008c00 records: 4, size/record: 24, table size: 104
.intvecs: algorithm=TMS570_CRC64_ISO(ID=10), load addr=00000000, size=00000020,
CRC=d9e7073eca088191
.text: algorithm=TMS570_CRC64_ISO(ID=10), load addr=00000020, size=00008730,
CRC=042202da2e5c3f1d
.const: algorithm=TMS570_CRC64_ISO(ID=10), load addr=00008750, size=00000288,
CRC=f93b08ae6214ed9b
.cinit: algorithm=TMS570_CRC64_ISO(ID=10), load addr=000089d8, size=00000228,
CRC=f7cf30240362a8da
```

### 3.3 Software Mode

The compression can be performed by the CPU with the following C function:

```
uint64 crc_update_word(uint64 crc64, uint64 data)
{
    int i, j;
    uint64 nextCrc = 0;

    // for i in 63 to 0 loop
    for(i = 63; i >= 0; i--)
    {
        // NEXT_CRC_VAL(0) := CRC_VAL(63) xor DATA(i);
        nextCrc = (nextCrc & 0xfffffffffffffeULL) | ((crc64 >> 63) ^ (data >> i));

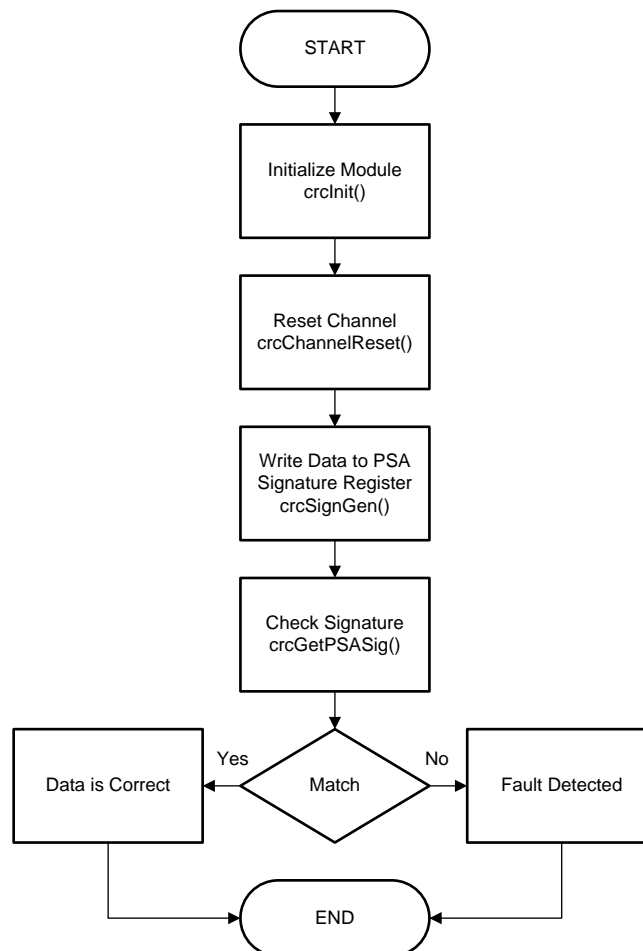
        // for j in 1 to 63 loop
        for(j = 1; j < 64; j++)
        {
            //case j is
            // when 1|3|4 =>
            if(j == 1 || j == 3 || j == 4)
            {
                // NEXT_CRC_VAL(j) := CRC_VAL(j - 1) xor CRC_VAL(63) xor DATA(i);
                nextCrc = (nextCrc & ~(1ULL << j)) | (((crc64 >> (j -
1)) ^ (crc64 >> 63) ^ (data >> i)) & 1) << j);
            }
            else
            { // when others =>
                // NEXT_CRC_VAL(j) := CRC_VAL(j - 1);
                nextCrc = (nextCrc & ~(1ULL << j)) | (((crc64 >> (j - 1)) & 1) << j);
            }
            // end case;
        } // end loop;
        crc64 = nextCrc;
    } // end loop

    return crc64;
}
```

The function has to be called for each 64-bit word, the inputs are the previous output or seed value (uint64 crc64) and the new data word (uint64 data). The function will not be very efficient when compiled as shown above, the performance can be improved by using proprietary compiler directives (#pragma). An improved version can be found in the CCS project provided with this document. The file containing the function is *CRC\_calc.c*.

### 3.4 Full-CPU Mode

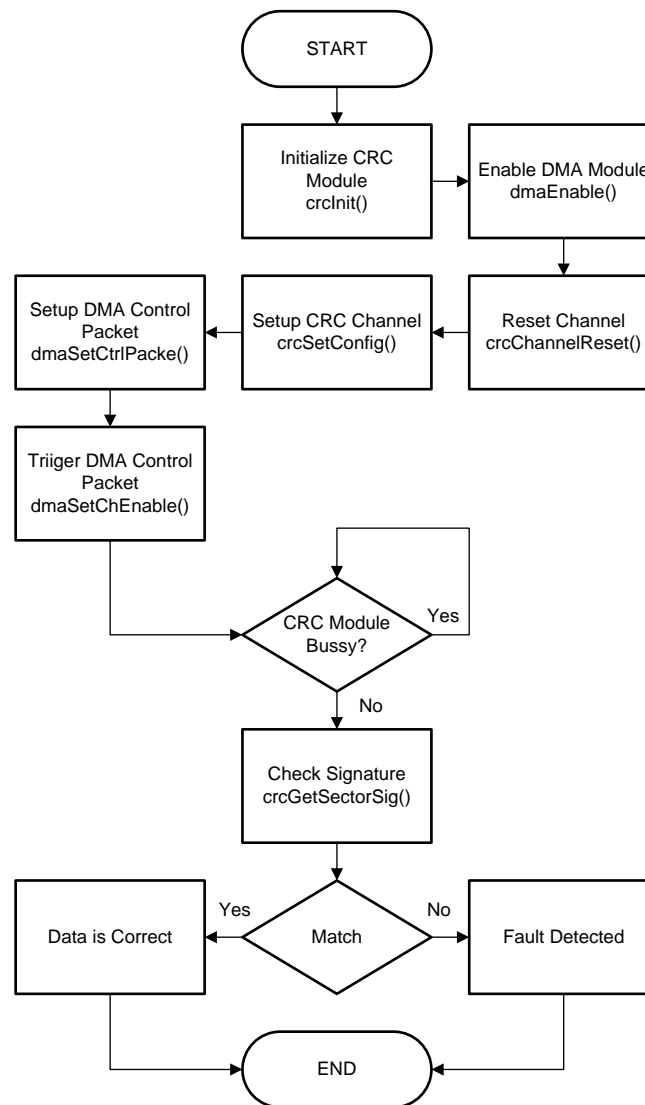
The full CPU-Mode example utilizes the function provided by HALCoGen to access the DMA module. In this mode, the CPU has to copy the data to be compressed into the CRC modules PSA Signature Register; this can be done with the help of the function *crcSignGen()* supplied with HALCoGen. After this, the CPU has to compare the newly calculated CRC with the pre-calculated CRC. Figure 3 shows the basic program flow.



**Figure 3. Full-CPU Mode With HALCoGen Flowchart**

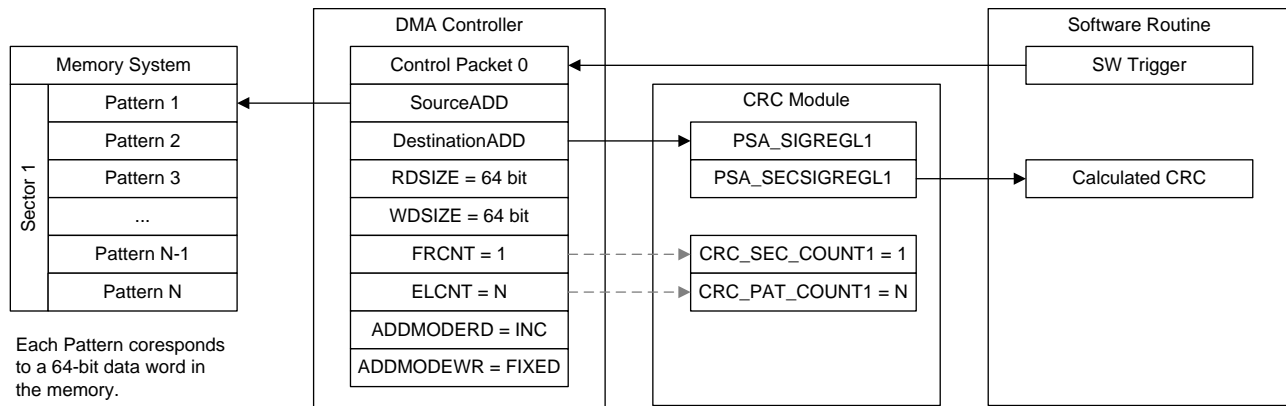
### 3.5 Semi-CPU Mode Polling

The Semi-CPU mode requires setting up a DMA packet, this can be done with the help of the DMA function supplied with HALCoGen. Similar to the Full-CPU Mode, HALCoGen generated functions are used to setup the CRC module. In addition to the CRC module setup used for the Full-CPU mode, the pattern and sector counters also have to be configured. The pattern counter is set to the number of double words (64 bit) to be compressed. The sector counter is set to 1 as only one sector at a time will be compressed. The time-out counters are used in this example. Figure 4 shows the program flow for this mode.



**Figure 4. Semi-CPU Mode Polling Flowchart**

The DMA control packet is setup by the application to copy the whole memory range to the CRC modules PSA Signature Register. The DMA will be used in software trigger mode, polling the status of the DMA control packet or the status of the CRC module can be used to determine if the data compression has been completed. [Figure 5](#) show the setup with the DMA controller.

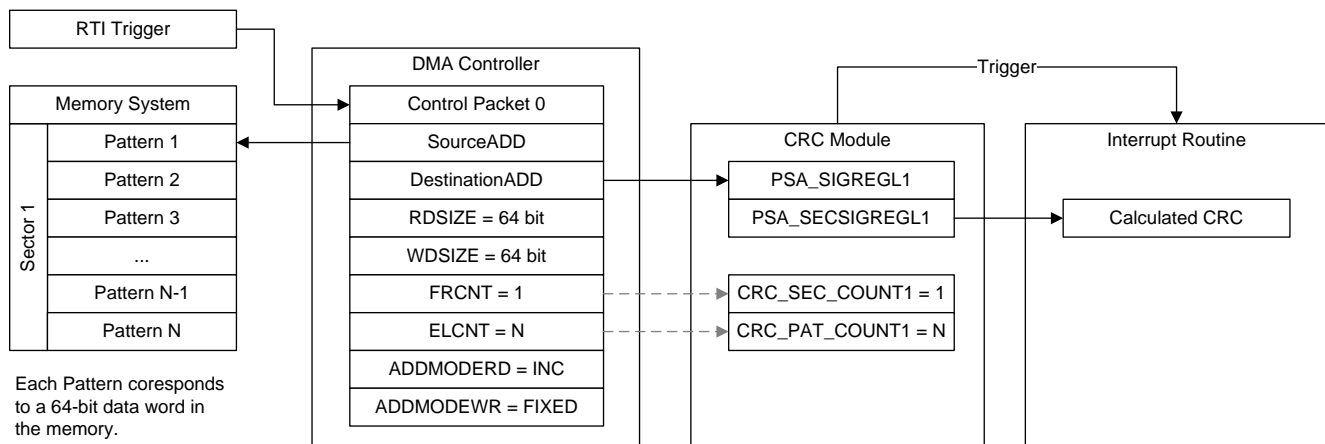


**Figure 5. Semi-CPU Mode Polling DMA Setup**

The DMA control packet is setup in a way that the DMA controller copies the memory area associated to a CRC table into the CRC modules PSA Signature Registers. The software can poll on the DMA's control packet status or the CRC modules Busy register to determine when the compression has been finished. The CRC modules Busy register is only functional if the Sector and Pattern count was setup in the CRC module. In that case, the CRC can be read from the PSA Sector Signature Registers.

### 3.6 Semi-CPU Mode Interrupt Driven

The setup for this mode is similar to the one from the Semi-CPU Mode Polling mode described in [Section 3.5](#). The main difference is that the DMA gets triggered by a timer (RTI) and the compression complete interrupt is used to tell the software that the current memory range has been compressed. [Figure 6](#) shows the setup with the DMA controller, RTI trigger and interrupt routine.



**Figure 6. Semi-CPU Mode Interrupt Driven Setup**

#### 3.6.1 Using the Time Out Counter

The trigger by the RTI can be used to issue individual 64-bit DMA read and writes. With a proper setup of the trigger frequency, the needed bandwidth can be adjusted to the applications requirements. Also, the duration for completing the compression of a memory range can be pre-calculated and set to the applications requirements. Knowing the expected duration for the compression allows to setup the timers in the CRC module to act as a watchdog for the DMA and signaling timeout events to the application in case the compression takes longer than expected (fault condition).

The timeout counter inside the CRC module is operating on  $HCLK / 64$ , which means  $200 \text{ MHz} / 64 = 3.125 \text{ MHz}$  - in case of the example provided for the RM48. The RTI compare 3 used in the same example operates on  $HCLK / 2 / 10 / 10 = 1 \text{ MHz}$  or  $1 \mu\text{s}$ . This means that the preload value (Block Complete Timeout Preload Register) has to be set at least to:

$$CRC_{BCTOPLDx} = \frac{CRC_{PCOUNT\_REGx} \times 10 \times 10 \times 2}{64} \quad (2)$$

Another way to calculate the timeout counter value is:

$$CRC_{BCTOPLDx} = \frac{CRC_{PCOUNT\_REGx} \times t_{RTI\_CMPx} \times f_{HCLK}}{64} \quad (3)$$

The example shown in [Section 3.6.2](#) demonstrates how to calculate the Block Complete Timeout Preload Register value.

### 3.6.2 Example, Sector Size = 32kB

CRC Channel 1 Pattern Counter Preload Register (note the sector size of 32kB):

$$CRC_{PCOUNT\_REG1} = \frac{32kB}{8 \text{ Byte}} = 4096 \text{ (64 bit words)} \quad (4)$$

CRC Channel 1 Block Complete Timeout Preload Register:

$$CRC_{BCTOPLD1} = \frac{CRC_{PCOUNT\_REG1} \times 10 \times 10 \times 2}{64} = \frac{4096 \times 10 \times 10 \times 2}{64} = 12800 \quad (5)$$

or

$$CRC_{BCTOPLD1} = \frac{CRC_{PCOUNT\_REG1} \times t_{RTI\_CMP3} \times f_{HCLK}}{64} = \frac{4096 \times 1\mu\text{s} \times 200\text{MHz}}{64} = 12800 \quad (6)$$

The value 12800 represents about 4 ms:

$$TimeOut = \frac{CRC_{BCTOPLD1} \times 64}{HCLK} = \frac{12800 \times 64}{200 \text{ MHz}} = 4 \text{ ms} \quad (7)$$

Compressing 32kB at a rate of 8 Byte (64-bit word) each microsecond will take:

$$TimeNeeded = \frac{SectorSize}{8 \text{ Byte}} \times t_{RTI\_CMP3} = \frac{32kB}{8 \text{ Byte}} \times 1\mu\text{s} = 4 \text{ ms} \quad (8)$$

## 4 Results and Conclusion

[Table 2](#) shows the measurement results for the RM48 example project (Example1: CRC Software Full-CPU and Semi-CPU Modes). It is visible that the use of the CRC module (see [Section 1.2.2](#)) offers a great performance boost compared to a software-based CRC implementation (see [Section 1.2.1](#), compare first two lines). The performance can be even more optimized by using the DMA instead of the CPU for copying the data from the Flash memory into the CRC module (compare lines two and three). It should also be noted, that the CPU is free (idle in the example) for doing other task most of the time shown in line three (see [Section 1.2.3](#)), this is also true for the interrupt driven mode.

**Table 2. Results Captured on RM48 (Example 1, 37kB)**

	CPU Cycles	Duration @ 200 MHz	Scaled to kB
<b>Software Mode</b>	23328899	116.6 ms	3152.6 us/kB
<b>Full-CPU Mode</b>	127604	0.6 ms	17.2 us/kB
<b>Semi-CPU Mode (Polling)</b>	47500	0.2 ms	6.4 us/kB

## IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

### Products

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>
OMAP Applications Processors	<a href="http://www.ti.com/omap">www.ti.com/omap</a>
Wireless Connectivity	<a href="http://www.ti.com/wirelessconnectivity">www.ti.com/wirelessconnectivity</a>

### Applications

Automotive and Transportation	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Communications and Telecom	<a href="http://www.ti.com/communications">www.ti.com/communications</a>
Computers and Peripherals	<a href="http://www.ti.com/computers">www.ti.com/computers</a>
Consumer Electronics	<a href="http://www.ti.com/consumer-apps">www.ti.com/consumer-apps</a>
Energy and Lighting	<a href="http://www.ti.com/energy">www.ti.com/energy</a>
Industrial	<a href="http://www.ti.com/industrial">www.ti.com/industrial</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Space, Avionics and Defense	<a href="http://www.ti.com/space-avionics-defense">www.ti.com/space-avionics-defense</a>
Video and Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>

### TI E2E Community

[e2e.ti.com](http://e2e.ti.com)