AN4187
Application note

Using the CRC peripheral in the STM32 family

Introduction

The cyclic redundancy check (CRC) is a technique used for detecting errors in digital data, but without making corrections when errors are detected. It is used in data transmission or data storage integrity check. The CRC is a powerful and easily implemented technique to obtain data reliability. Diagnostic coverage of this technique satisfies requirements of basic safety standards. This is why the CRC implementation feature is used at Flash content integrity self-test check at ST firmware certified for compliance with IEC 60335-1 and IEC 607030-1 standards (known as "Class B" requirements). For more information, refer to application note AN3307 and the associated firmware packages dedicated for different family products. It is advised to check all the necessary CRC settings at compilers' manuals when CRC checksum information has to be placed directly into user code by linker (mostly in format of a CRC descriptor data table).

The CRC is based on polynomial arithmetic. It computes the remainder of the division of a polynomial in GF(2) by another. The remainder is called checksum, while the dividend is the data and the divisor is the generator polynomial.

Note: A polynomial in GF(2) (Galois field with two elements) is a polynomial with a single variable x whose coefficients are 0 or 1.

This application note describes the features of the cyclic redundancy check peripheral embedded in all STM32 series (F0, F1, F2, F3, F4, L1) and the steps required to configure it.

This application note is structured as follows:

- *Section 1* describes the STM32 CRC implementation algorithm and its hardware implementation benefits.
- *Section 2* describes the use of the DMA as CRC data transfer controller.
- *Section 3* describes the migration of the CRC through STM32 devices.

Two examples are provided as well:

- CRC_usage example: how to configure the CRC using CPU as data transfer controller.
- CRC_DMA example: how to use the DMA as CRC data transfer controller.

The measuring of execution time in both examples is supported.

Table 1. Applicable products

Type	Product category
Microcontrollers	STM32

Contents

List of tables

List of figures

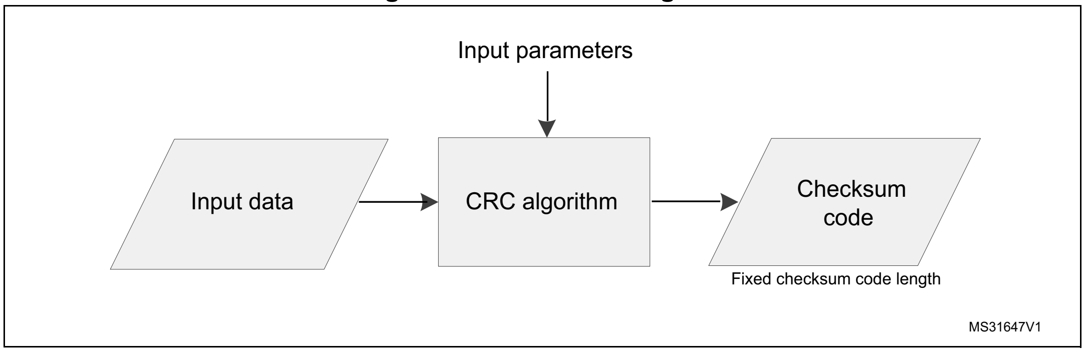
1 CRC peripheral overview

The CRC peripheral embedded in all STM32 microcontroller devices is used to provide a CRC checksum code of any supported data type.

1.1 CRC computing algorithm

As shown in *Figure 1*, the CRC algorithm has a data input and generates a fixed checksum code length depending on the input parameters.

Figure 1. CRC block diagram



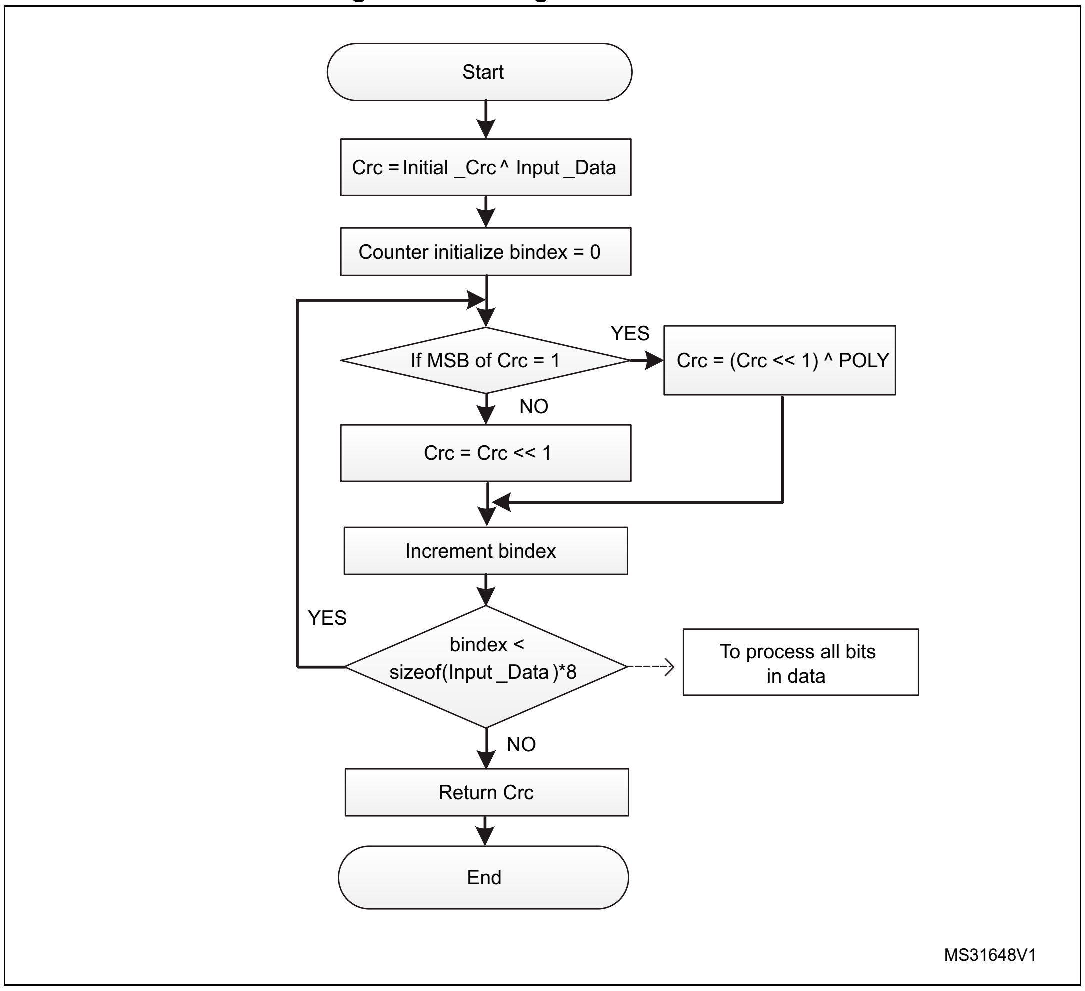
One of the known CRC algorithms is the polynomial division with the bitwise message XORing technique. This is the most suitable technique for hardware or low-level implementation, as used in microcontrollers.

The input parameters of this algorithm are:

- the dividend: also called input data, abbreviated to *"Input_Data"*
- the divisor: also called generator polynomial, abbreviated to "*POLY*"
- an initial CRC value: abbreviated to "*Initial_Crc*"

Figure 2 below shows the CRC algorithm flowchart.

Figure 2. CRC algorithm flowchart


Start
Crc = Initial _Crc ^ Input _Data
Counter initialize bindex = 0
If MSB of Crc = 1
YES
Crc = (Crc << 1) ^ POLY
NO
Crc = Crc << 1
Increment bindex
YES
bindex < sizeof(Input _Data)*8
To process all bits in data
NO
Return Crc
End
MS31648V1

At start up, the algorithm sets CRC to the *Initial_Crc* XOR with the *Input_Data.*

Once CRC MSB is equal to one, the algorithm shifts CRC one bit to the left and XORs it with the POLY. Otherwise, it only shifts CRC one bit to the left.

Figure 3 shows the step-by-step algorithm execution for the following conditions:

- Input_Data = 0xC1
- POLY = 0xCB
- Initial_Crc = 0xFF

Figure 3. Step-by-step CRC computing example

bindex	Execution step	Binary format	Hex
	Crc = *Initial_Crc* ^ *Input_Data*	1 1 1 1 1 1 1 1 (*Initial_Crc*)	0xFF
		^ 1 1 0 0 0 0 0 1 (*Input_Data*)	0xC1
0	Crc << 1	$\boxed{0}$0 1 1 1 1 1 0	0x3E
1	Crc << 1	$\boxed{0}$1 1 1 1 1 0 0	0x7C
2	Crc << 1	$\boxed{1}$1 1 1 1 0 0 0	0xF8
	Crc = Crc ^ POLY	$\boxed{1}$1 1 1 0 0 0 0	0xF0
		^ 1 1 0 0 1 0 1 1 (*POLY*)	0xCB
3	Crc << 1	$\boxed{0}$0 1 1 1 0 1 1	0x3B
4	Crc << 1	0 1 1 1 0 1 1 0	0x76
5	Crc << 1	$\boxed{1}$1 1 0 1 1 0 0	0xEC
	Crc = Crc ^ POLY	$\boxed{1}$1 0 1 1 0 0 0	0xD8
		^ 1 1 0 0 1 0 1 1 (*POLY*)	0xCB
6	Crc << 1	$\boxed{0}$0 0 1 0 0 1 1	0x13
7	Crc << 1	$\boxed{0}$0 1 0 0 1 1 0	0x26
	Crc (Returned value)	0 1 0 0 1 1 0 0	0x4C

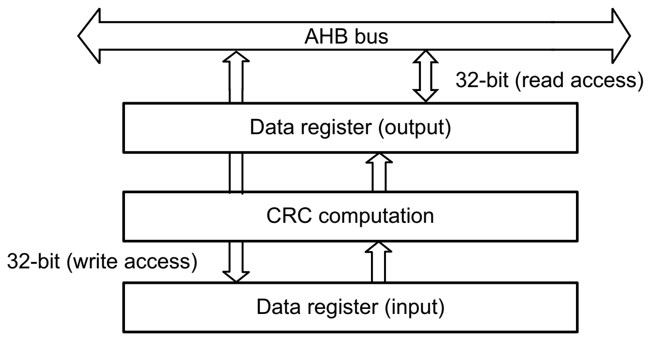
1. The returned CRC value (0x4C) has been verified by the CRC peripheral in STM32F37x family with the configurations mentioned above.
2. This routine has been implemented under CrcSoftwareFunc function in CRC_usage example in software package.

1.2 CRC peripheral configuration

All STM32 devices implement a CRC peripheral as described in *Section 1.1*. The CRC calculation unit has a single 32-bit read/write data register (CRC_DR). It is used to input new data (write access) and hold the result of the previous CRC calculation (read access).

Each write operation to the data register creates a combination of the previous CRC value (stored in CRC_DR) and the new one.

Figure 4. CRC calculation unit block diagram



MS19882V1

To compute a CRC of any supported data, you must follow these steps:

1. Enable the CRC peripheral clock via the RCC peripheral.
2. Set the CRC Data Register to the initial CRC value by configuring the Initial CRC value register (CRC_INIT).[a]
3. Set the I/O reverse bit order through the REV_IN[1:0] and REV_OUT bits respectively in CRC Control register (CRC_CR).[a]
4. Set the polynomial size and coefficients through the POLYSIZE[1:0] bits in CRC Control register (CRC_CR) and CRC Polynomial register (CRC_POL) respectively.[b]
5. Reset the CRC peripheral through the Reset bit in CRC Control register (CRC_CR).
6. Set the data to the CRC Data register.
7. Read the content of the CRC Data register.
8. Disable the CRC peripheral clock.

In firmware package, the CRC_usage example runs the CRC checksum code computing an array data (*DataBuffer*) of 256 supported data type. For a full description, please refer to the file *Readme.txt* in the CRC_usage folder.

a. Applicable only for STM32F0xx and STM32F3xx devices

b. Applicable only for STM32F3xx devices

1.3 CRC hardware implementation benefits

The CRC_usage example has been developed to check the compatibility between the software CRC algorithm implementation and the CRC peripheral, as well as to measure their execution times.

The example has been executed under the following conditions:

– Hardware: STM32373C-EVAL board (STM32F37x device)
– System clock: HSE (8 MHz crystal oscillator)
– Toolchain: Keil V4.60.0.0
– CRC configurations: Default values of the CRC registers
  CRC_CR: 0x0000 0000; POLYSIZE is 32, No REV_IN and No REV_OUT
  CRC_INIT: 0XFFFF FFFF
  CRC_POLY: 0X04D11 CDB7
– Input data: 256 words

Table 2 shows the results of the comparison of the execution time of the CRC algorithm versus the STM32 CRC peripheral.

Table 2. Comparison of CRC algorithm and CRC peripheral execution time

Optimization level	CRC algorithm (system clock cycle)	CRC peripheral (system clock cycle)
Level 3 + Optimize for time	78094	1287

The hardware implementation of the CRC is about 60 times faster than the software algorithm. The CPU controls the data transfer and no instruction process is allowed during this phase. Application developers can choose another peripheral as controller in order to free the CPU for other tasks. Since the DMA manages data transfer, it becomes the alternative choice for computing the CRC checksum code of the data buffer.

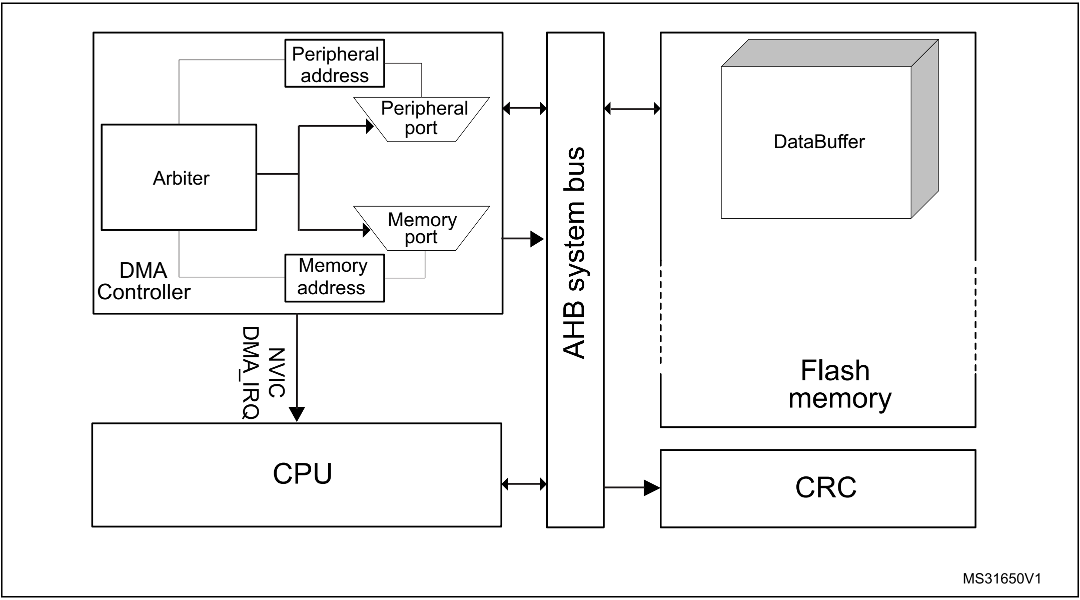
2 Using CRC through DMA

The STM32 embedded DMA can be used as the data transfer handler to avoid the use of the CPU as controller. The DMA configuration depends on the architecture chosen. there are two categories: DMA with channels and DMA with stream. For both DMA architectures, the configuration is almost identical.

2.1 DMA back-to-back transfer mode

The CRC_DMA example available in the firmware package implements the technique illustrated in *Figure 5*. This technique is the DMA back-to-back data transfer mode with the DMA_IRQ handler routine callback.

Figure 5. CRC computing through DMA transfer



In this case of use, the DMA controls the data transfer counter and waits for the transfer complete flag to execute the NVIC DMA_IRQ handler routine. The NVIC DMA_IRQ routine has to stop the system timer (systick) counter and return the CRC computed value. The CPU usage is only limited to the execution of the DMA interrupt request routine.

2.2 DMA configuration

As mentioned above, STM32 devices integrate a multiple DMA architecture. The configuration steps below are common for both DMA architectures:

1. Enable the DMA peripheral clock via the RCC peripheral.
2. Configure the DMA channel/stream (see *Table 3* for the two configurations).
3. Configure the CRC, as described in the first five steps of *Section 1.2*.
4. Enable the DMA transfer complete interrupt.
5. Configure the DMA NVIC IRQ.
6. Enable the DMA channel/stream.
7. Wait for the DMA transfer complete to occur.
8. Disable the DMA channel. In the case of DMA with stream architecture, the controller disables automatically the channel when the transfer ends, while in the other case of architecture, the NVIC DMA_IRQ routine must disable the channel for further use.
9. Clear DMA IT pending bit.

Note: For any additional information please refer to the file Readme.txt under CRC_DMA example folder.

Table 3. DMA configuration summary

DMA configuration	DMA with channel	DMA with stream
Transfer direction	Memory to Memory	
Peripheral address	Flash Base pointer	
Memory address	CRC data register	
Peripheral address increment	Enable	
Memory address increment	Disable	
Buffer size	Data Buffer size	
Peripheral data size	Supported data type (byte, half-word or word)	
Memory data size	Supported data type (byte, half-word or word)	
Transfer Mode	Normal	
Peripheral Burst	NA	Single
Memory Burst	NA	Single
FIFO mode(1)	NA	Disable
FIFO threshold(1)	NA	--

1. Enabling the FIFO mode does not affect the execution time. Therefore, the FIFO mode and the FIFO threshold configuration have no impact.

The CRC_DMA example has been developed to check the compatibility results between the use of CPU and DMA as transfer handlers, and measure the CPU load during the use of DMA as transfer handler.

2.3 DMA usage benefits

During the DMA back-to-back data transfer mode, the CPU intervenes only during the CRC and DMA configurations and during the DMA interrupt handler execution.

The execution conditions of the example are the same as those listed in *Section 1.3*, except for the input data size that became 8192 of the supported data type.

Table 4. Comparison results of CPU versus DMA execution time usage

Transfer handler	Peripheral configuration(1)	CRC computing(1)	CPU load
CPU	66(2)	40962	100%
DMA	295(3)	40968	0.72%

1. The systick timer tick is the measurement unit, while the systick clock source is the CPU clock.
2. CRC configuration time
3. CRC configuration, DMA configuration and DMA IRQ handler execution times

Overall, it is important to note that the CPU load is equal to 100% when the CPU is used as the data transfer handler, while it is reduced to 0.72 % when using DMA.

3 CRC migration through STM32 series

The CRC peripheral features can vary from one STM32 series to another. *Table 5* lists the CRC features and offers a software compatibility analysis for each STM32 series.

Table 5. STM32 CRC peripheral features

Feature	F1 series	L1 series	F2 series	F4 series	F0 series	F3 series
Single input/output 32-bit data register	YES	YES	YES	YES	YES	YES
General-purpose 8-bit register	YES	YES	YES	YES	YES	YES
Input buffer to avoid bus stall during calculation	NO	NO	NO	NO	YES	YES
Reversibility option on I/O data	NO	NO	NO	NO	YES	YES
CRC initial value	Fixed to 0XFFFFFFFF	Fixed to 0XFFFFFFFF	Fixed to 0XFFFFFFFF	Fixed to 0XFFFFFFFF	Programmable on 32 bits	Programmable on 8, 16, 32 bits
Handled data size in bits	32	32	32	32	32	8, 16, 32
Polynomial size in bits	32	32	32	32	32	7, 8, 16, 32
Polynomial coefficients	Fixed to 0x4C11DB7	Fixed to 0x4C11DB7	Fixed to 0x4C11DB7	Fixed to 0x4C11DB7	Fixed to 0x4C11DB7	Programmable

4 Reference documents

- STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MCUs reference manual (RM0008)
- STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx advanced ARM-based 32-bit MCUs reference manual (RM0033)
- STM32L151xx, STM32L152xx and STM32L162xx advanced ARM-based 32-bit MCUs reference manual (RM0038)
- STM32F100xx advanced ARM-based 32-bit MCUs reference manual (RM0041)
- STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs reference manual (RM0090)
- STM32F05xxx advanced ARM-based 32-bit MCUs reference manual (RM0091)
- STM32F37xx and STM32F38xx advanced ARM-based 32-bit MCUs reference manual (RM00313)
- STM32F302xx, STM32F303xx and STM32F313xx advanced ARM-based 32-bit MCUs reference manual (RM00316)
- Guidelines for obtaining IEC 60335 Class B certification in STM32 applications (AN3307)

Note: The above documents are available at the following URL: http://www.st.com/stm32

5 Revision history

Table 6. Document revision history

Date	Revision	Changes
06-Jun-2013	1	Initial release.