

# Exam: embedded electronics (M1, semester 1)

J.-M Friedt, December 19, 2025

2 points/correct answer, negative points for erroneous answers copied from search engines or chatbots.

We aim at understanding the Cyclic Redundancy Check (CRC) computation for detecting messages corrupted during transmission, whether over wired or wireless communication links. With over 1.5 zettabytes ( $1.5 \cdot 10^{15}$ ) shared in 2025 over the Internet<sup>1</sup>, this is arguably one of the most commonly executed algorithms, being needed to check that each and every byte was received unchanged from its emission.

CRC calculations are very efficient to identify whether one or multiple bits have been corrupted between emitter and receiver, and are ubiquitous to most modern digital communication schemes. For example, each Ethernet frame with a payload size ranging from 42 to 1500 bytes (or larger when using jumbo frames) ends with a 32-bit CRC allowing to identify whether some of the information has been corrupted during transmission. CRC is however unable to identify the location of the corrupted bit(s) and recover the information, which is taken care of by Forward Error Correction (FEC) schemes which are beyond the topic of this exam.

1. What deficiency of parity bit, found for example in RS232, could a CRC address that the parity bit cannot?

The STM32F103 we have been using is able to compute CRC checksums thanks to some dedicated hardware peripheral. CRC checksums are ubiquitous in all modern communication systems and implemented as follows<sup>2</sup> in order to encode a bit input stream with a primitive polynomial  $p$  defined with as many bits plus 1 as the CRC length. The leading bit of  $p$  is always 1 and will be implicit from now on:

- initialize a CRC register with the number of bits expected from the output checksum with a known pattern, most often all 0s or all 1s
  - shift the input bitstream into the least significant bit side of the CRC register
  - if the most significant bit of the CRC register is 1, XOR with  $p$ , otherwise continue shifting until the most significant bit becomes 1.
  - repeat for all the input bits until the stream is exhausted: the resulting CRC register value is the wanted checksum.
2. Hardware CRC computation is supported on the STM32F103: provide the register addresses involved in the computation. What are the functions of these registers? Where did you find the description of these registers, and their usage?
  3. Can the STM32F103 compute the CRC used on CD-ROM mass storage medium defined with the primitive polynomial  $p=0x8001801B$ ?

The website <https://crccalc.com> does provide the result of CRC computation for some given input pattern, or for the `Check` value made of the ASCII string made of byte concatenation "123456789". Notice that the output CRC can possibly be XORed with a pattern provided as the `XorOut` field on this web site (right-most column).

4. Add to your STM32 library a new function allowing to compute the CRC of a 32-bit integer message made of the four bytes 0x00, 0x01, 0x02 and 0x03. Does libopenm3 support the STM32F1 CRC computation? If yes, how can you access these resources? If no, how did you implement the hardware CRC computation?
5. What peripheral of a STM32 hardware function should you never forget to initialize so it operates properly? What happened in this case if we had forgotten this initialization for CRC calculation?

---

<sup>1</sup><https://www.itu.int/itu-d/reports/statistics/2025/10/15/ff25-internet-traffic/>

<sup>2</sup>[https://www.sunshine2k.de/articles/coding/crc/understanding\\_crc.html#ch3](https://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#ch3)

6. Compare your result, executed on the STM32F103, with those provided by <https://crccalc.com>. Based on the **Reset** state of the CRC register of the STM32F1, what CRC is computed according to the web site table? How does your result compare?
7. Are you able to compute the **Check** quantity using the STM32F1 hardware? If yes, describe the procedure and the result. If not, why?

We wish to implement manually the CRC calculation on the STM32F1. The CRC calculation on an byte input array `in` with the 32-bit primitive polynomial `p` is provided in section 6 of <sup>3</sup>.

8. adapt this example to C and execute on the PC: check whether the output matches the expectation using <https://crccalc.com/?method=CRC-32> for the input bytes we used earlier and for the **Check** input array.
9. How long does your manual CRC implementation need to execute? How long does the hardware implementation need to execute? Compare.
10. Had we wanted to compute the CD-ROM checksum, what alternative to software CRC implementation could we have selected?

---

<sup>3</sup>[https://www.sunshine2k.de/articles/coding/crc/understanding\\_crc.html#ch6](https://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#ch6)

## Solutions

1. The parity bit can only detect an odd number of bit corruption and fails to detect an even number of bit change during communication. The CRC has a much higher chance of detecting multiple bit changes.
2. Reference Manual 0008 section 4: data register CRC\_DR at base+0x00, CRC\_IDR at base+0x04 and control register CRC\_CR at base address+0x08 with the base address of all CRC related registers locate at 0x40023000.
3. The STM32F103 can only compute the CRC for primitive polynomial 0x4C11DB7 as used in Ethernet (section 4.2 of the manual) and is unable to compute the CD-ROM checksum requiring the primitive polynomial 0x8001801B.
4. the following piece of simplified software (missing general clock, GPIO clock and UART clock initialization) uses the libopencm3 functions defined in include/libopencm3/stm32/f1/crc.h to compute the CRC using the hardware peripheral:

```
#include "libopencm3/cm3/common.h" // BEGIN_DECLS
#include "libopencm3/stm32/f1/crc.h" // find crc.h
#include "libopencm3/stm32/f1/rcc.h" // crc clock

int main()
{char c[4]={0x1,0x2,0x3,0x4};
 long *p=(long*)c;
 rcc_periph_clock_enable(RCC_CRC);
 while(1)
 {crc_reset();
  res=crc_calculate(*p);
  res=res^0xffffffff;
  my_long(res);
 }
 return 0;
}
```

5. the clock of every STM32 peripheral must be initialized (rcc\_periph\_clock\_enable(RCC\_CRC);) otherwise the output is 0 (at least the CRC calculator does not hang the execution)
6. Selecting HEX input and using the input stream 01 02 03 04 in <https://crccalc.com> indeed provides the CRC-32/BZIP2 (initialization state 0xFFFFFFFF) the result 0x86C8C832 after XORing the output with 0xFFFFFFFF.

Note: our attention was brought to the fact that the endianness of the bytes has not been defined, and that the sequence 03 02 01 00 does match the CRC32/MPEG-2 CRC without XORing the output. This CRC does match the initial state of all 0xFFFFFFFF shift register.

7. The STM32F1 can only compute the CRC on 32-bit long inputs while the **Check** sentence is 9 byte long. We have not been able to compute the **Check** output using the hardware CRC implementation in the STM32F103.
8. The following program answers the question on the PC:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

uint32_t Compute_CRC32_Simple(char *bytes,int len,uint32_t init_crc)
{const uint32_t poly=0x04C11DB7; // 32 bit divisor, implicit leading 1
 uint32_t crc=init_crc; // CRC value is 32bit
 int i,k;
 for (k=0;k<len;k++) // move byte into MSB of 32bit CRC
 {crc^=(uint32_t)(bytes[k]<<24);
  for (i=0;i<8;i++) // vv test for MSB = bit 31
   if ((crc&0x80000000)!=0) crc=(uint32_t)((crc<<1)^poly);
   else crc<<=1;
 }
}
```

```

    return crc;
}

int main()
{char in1[9]={'1','2','3','4','5','6','7','8','9'};
 char in2[4]={0x01,0x02,0x03,0x04};
 uint32_t crc;
 crc=Compute_CRC32_Simple(in1,9,0x00000000);
 printf("CKSUM:_%08x->_%08x\n",crc,crc^0xffffffff);
 crc=Compute_CRC32_Simple(in1,9,0xffffffff);
 printf("BZIP2:_%08x->_%08x\n",crc,crc^0xffffffff);
 crc=Compute_CRC32_Simple(in2,4,0x00000000);
 printf("\nCKSUM:_%08x->_%08x\n",crc,crc^0xffffffff);
 crc=Compute_CRC32_Simple(in2,4,0xffffffff);
 printf("BZIP2:_%08x->_%08x\n",crc,crc^0xffffffff);
}

```

with the output

```

CKSUM: 89a1897f -> 765e7680
BZIP2: 0376e6e7 -> fc891918

```

```

CKSUM: be33eab6 -> 41cc1549
BZIP2: 793737cd -> 86c8c832

```

matching the output of <https://crccalc.com/> for both the CRC-32/CKSUM (initialization to all 0s) and CRC-32/BZIP2 (initialization to all 1s).

9. Toggle a GPIO and measure on the oscilloscope the execution duration of both hardware and software implementation on the STM32, making sure to remove the communication over the asynchronous serial line which takes much more time than the computation itself. Using the following program

```

#define usart1
#include "uart.h"
#include <stdint.h>
#include "libopencm3/cm3/common.h" // problem de BEGIN_DECLS
#include "libopencm3/stm32/f1/crc.h" // trouver crc.h
#include "libopencm3/stm32/f1/rcc.h" // crc clock

#define N 4

uint32_t Compute_CRC32_Simple(char *bytes,int len,uint32_t init_crc)
{[... see above ...]
}

int main()
{int msk;
 char d[N]={0x04,0x03,0x02,0x01}; // little endian
 uint32_t *d32=(uint32_t*)d;
 uint32_t res;
 msk=(1<<11)|(1<<12)|(1<<13);
 clock_setup();
 init_gpio();
 usart_setup();

 rcc_periph_clock_enable(RCC_CRC);
 while(1)
 {my_long(*d32);
 led_set(msk);
 crc_reset(); res=crc_calculate(*d32); // using : 0x4C11DB7
 led_clr(msk);
 my_putchar('>');my_putchar('<');
 my_long(res^0xffffffff);
 led_set(msk);
 res=Compute_CRC32_Simple(d,4,0xffffffff);
 led_clr(msk);
 my_putchar('_');
 }
}

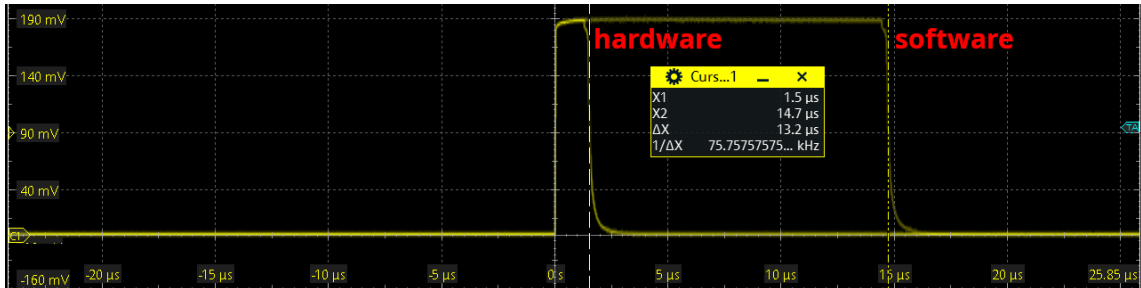
```

```

    my_long(res^(0xffffffff));
    my_putchar('\r'); my_putchar('\n');
    delay(0xffff);
}
return 0;
}

```

we observe on the oscilloscope that the hardware solution is 10 times faster than the software solution:



with the `crc_reset()` function accounting for about  $0.4 \mu\text{s}$  of the global  $1.5 \mu\text{s}$  of the hardware solution.

10. Newer models of the STM32 provide a more flexible CRC hardware calculation peripheral where the CRC length and primitive polynomial can be defined through the `POLYSIZE` and `CRC_POLY` registers, missing in the STM32F103. See ST Microelectronic's AN4187 *Using the CRC peripheral on STM32 microcontrollers* (2022) which also concludes that the hardware version is 60 times faster than the software implementation.

Notice that surprisingly enough, when reaching the STM32 registers directly and without using `libopen3`, the code

```

CRC_CR |= CRC_CR_RESET;
__asm__("nop");
CRC_DR = 0x00010203;
return CRC_DR;

```

returns a CRC calculation, but

```

CRC_CR |= CRC_CR_RESET;
CRC_DR = 0x00010203;
return CRC_DR;

```

always returns `0xffffffff`. This mandatory delay between reset and calculation does not seem to be documented in the reference manual.