# A Fast Compact CRC5 Checker For Microcontrollers

Dipl.-Ing. Michael Joost
Research and Development, 47829 Krefeld, Germany, EU

***Abstract*—This paper proposes a fast and compact software implementation for checking the validity of Cyclic Redundancy Codes of order five (CRC5), such as used in the USB protocol. The algorithm is adapted to the memory- and clock cycle restrictions found in small microcontrollers.**

## I. Introduction

Cyclic Redundancy Codes (CRC) are commonly used for detecting payload corruption arising from transmission errors [1]. In particular, the Universal Serial Bus (USB [2, sect. 8]) uses CRC16 for securing payload data and CRC5 for securing address/endpoint information in SETUP/IN/OUT token packets. We will focus on the latter in this paper.

Efficient software implementations of CRC calculations have been known for a long time, usually deploying a lookup table using a 8-bit-wise processing of the payload (see [3]). However, the size of the byte-wise lookup table (256 entries of one byte each for CRC5) puts a rather significant memory footprint for small microcontrollers controlling a device. Bitwise algorithms avoid the memory required for the lookup-table at the expense of excessively more clock cycles. For microcontrollers supporting nibble operations, a table lookup of significantly smaller size with two lookup operations per byte can be implemented.

In USB the address/endpoint information is eleven bits wide, and secured by the five bit wide CRC. Since the validity checking is accomplished by calculating the CRC over both these fields, and comparing to a known residual, our algorithm can operate on an integral number of bytes, in particular, two bytes.

## II. CRC Algebra

A CRC algorithm treats the bits of a payload data of length $l$ bits, padded by appending $g$ zero bits, as the coefficients of a binary polynomial $B(x)$ of degree $b = l + g - 1$, and divides that by a constant generator polynomial $G(x)$ of degree $g$. Appending the padding bits reserves space for adding the CRC lateron, and enforces all payload bits up to the least-significant bit (LSB) being mangled thru the division.
The quotient $Q(x)$ of the division is not relevant. The remainder

$$R(x) \;=\; B(x) \bmod G(x)$$

of this operation, a polynomial of degree $r = g-1$, is the CRC that is appended to the payload data, replacing the padded zero-bits:

$$\begin{aligned} & B(x) && = Q(x) \cdot G(x) + R(x) \\ \Leftrightarrow \quad & \underbrace{B(x) - R(x)}_{\text{message}} && = Q(x) \cdot G(x) \end{aligned}$$

Since the message (payload+CRC) is always divisible by the generator polynomial $G(x)$, we can check validity by calculating the CRC on the entire message, which should leave a zero remainder. This is enabled by the appending of the zero bits when originally calculating the CRC.

The division is calculated on the finite Galois field of two elements, GF(2). In this finite field the operations are carry-less, so the field's addition and subtraction operations degenerate to the exclusive-OR (XOR) of just the corresponding elements (bit-wise).

The generator polynomial for CRC5 used in USB is (see [2, sect 8.3.5.1])

$$\begin{aligned} G(x) &= x^5 + x^2 + x^0 \qquad \text{of degree } g = 5 \\ &= 1 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 \end{aligned}$$

This generator polynomial can therefore be interpreted as a bitstring of six bits length: $100101$.

### A. CRC Compostion Operation

The division is a linear, distributive operation, so

$$(x \oplus y) \bmod G = (x \bmod G) \oplus (y \bmod G)$$

or

$$crc(x \oplus y) = crc(x) \oplus crc(y)$$

where $\oplus$ denotes the bit-wise XOR operation.

However, many real-world CRC functions add a constant to the remainder, as we will discuss soon. In this case the above equation does not hold:

$$\begin{aligned} CRC(x) &= (x \bmod G) \oplus c \qquad \text{with } c \neq 0 \\ CRC(x \oplus y) &= ((x \oplus y) \bmod G) \oplus c \\ &= (x \bmod G) \oplus (y \bmod G) \oplus c \\ &\neq ((x \bmod G) \oplus c) \oplus ((y \bmod G) \oplus c) \\ &= CRC(x) \oplus CRC(y) \end{aligned}$$

http://www.michael-joost.de/tech.html

### B. Bit Sequence Operation

Assume a bit sequence consisting of two parts, a prefix sequence $P$ and a suffix sequence $S$. The partial CRC over $P$ is already known as

$$R_p = P \bmod G \quad \Rightarrow \quad P \cdot 2^g = Q_p G + R_p$$

The CRC over the sequence $B(x) = [P(x) : S(x)]$ is calculated as

$$R = [P : S] \bmod G = (P \cdot 2^{slen} \oplus S) \bmod G$$

where $slen$ denotes the length of $S$ in bits.

Substituting the partial CRC we get

$$\begin{aligned} R &= ((Q_p G \oplus R_p) \cdot 2^{slen} \oplus S) \bmod G \\ &= ((Q_p G \cdot 2^{slen}) \oplus (R_p \cdot 2^{slen} \oplus S)) \bmod G \\ &= ((Q_p G \cdot 2^{slen}) \bmod G) \oplus ((R_p \cdot 2^{slen} \oplus S) \bmod G) \\ &= (R_p \cdot 2^{slen} \oplus S) \bmod G \quad = \quad [R_p : S] \bmod G \end{aligned}$$

We find that $crc([P : S]) = crc([crc(P) : S])$. The prefix sequence can be substituted for its partial CRC residual without changing the overall CRC residual.

### C. An Example

To clarify on the long division used in the CRC caclculation we consider an **Example.**
We consider the 11-bit payload string 10100111010 (53A). By padding five zero bits at the end we get the polynomial $B(x)$ of degree 15:

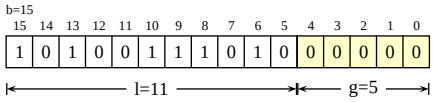


Fig. 1. Message Example

The polynomial division of that message string by the CRC5 polynomial on GF(2) can be evaluated manually much the same as in conventional math. We start with the current remainder initialized to the first $g + 1$ bits of the divisor. Whenever the most significant bit of the current remainder is 1, we subtract (using modulo-2 operation) the generator polynomial from the remainder. Then any leading zero-bits are discarded from the remainder, and a corresponding number of bits is fed from the dividend to the right end of the current remainder. The calculation stops when no further bits are left from the dividend.

```
 10100111010 00000    mod 100101
-100101 |         |
 ------ |         |
  0110011         |
  -100101         |
   ------         |
    101100        |
   -100101        |
    ------        |
     0100110      |
     -100101      |
      ------      |
       00011 0000|
         -10 0101|
          -- ----|
           1 01010
          -1 00101
           - -----
             01111    remainder
 10100111010 01111    message
```

A problem with this simple approach is that any leading zero-bits in the payload do not affect the remainder, thus are not protected. Inserted zero-bits at the beginning could not be detected, as well as a replacement by zero of both, the payload and the CRC (a long burst error). Therefore, typical CRC algorithms demand that a constant value is added (modulo-2) to the head of the payload message. This is usually a bit-string of all-1 bits with the size of the CRC polynomial's degree, effectively inverting the first $g$ bits of the payload. In case of CRC5 in USB this is defined as the 5-bit string 11111. Likewise, the USB standard demands that the remainder is inverted when inserted into the message, so that trailing zero-bits in the payload are protected.

As a consequence of these modifications, the division of the complete message (payload+CRC) by the generator polynomial no longer leaves a zero remainder. Both these operations have the effect of adding a constant (though depending on the message length) value to the division's remainder.
Furthermore, the USB standard demands that all data is sent with the least-significant bit first. An interesting side effect of the carry-less modulo-2 operation on GF(2) is that reversing the payload (but not the generator polynomial) results in the same, but reversed, remainder value from the polynomial division. Therefore, we can neglect the bit-order dependencies from our considerations.

Incorporating the above modifications now gives the correct results as expected by the USB standard:

```
 10100111010 00000    mod 100101
 11111               |
 -----               |
  101111             |
 -100101             |
  ------             |
   0101010           |
  -100101            |
   ------            |
    0111110          |
   -100101           |
    ------           |
     11011 0         |
    -10010 1         |
     ----- -         |
      1001 10        |
     -1001 01        |
      ---- --        |
       000 11000     |  remainder
           -----     |
     inv: 00111      |  CRC
```

For checking the validity of a received message we use the exact *same* mechanism as in calculating the CRC at the sender, just on a longer 'payload' field of 16 bit, comprising the original payload and the appended CRC. That also includes inverting the top-most five bits and appending five zero-bits.
Appending the zero-bits isn't strictly necessary for validity checking. The reason is just to use the same algorithm as for generating the CRC, and to retrieve the same residual as defined in the USB standard. We could also check for a remainder of 11111 after processing the last bit of the CRC. The effect of processing the subsequent trailing zero-bits is independent of the payload and the CRC, thus, results always in the same values.

```
10100111010 00111 00000    mod 100101
11111
-----
0101111
-100101
 ------
  0101010
  -100101
   ------
    0111110
    -100101
     ------
      11011 0
      -10010 1
       ----- -
        1001 10
        -1001 01
         ---- --
          000 11111 0    if CRC valid, this is 11111
              -10010 1
               ----- -
                1101 10
                -1001 01
                 ---- --
                  100 110
                  -100 101
                   --- ---
                    00 01100
                       01100    remainder
```

This is the residual specified in the USB standard for any CRC-valid message.

## III. Optimizations

### A. Tail-Cutting

As indicated before, the processing of the trailing zero-bits in validity checking isn't really necessary. Also, when calculating the CRC, the final value is only dependent on the remainder achieved after processing the last payload bit. This gives reason to have a closer look at the XOR-and-shift operations we do in the long division:

Check the most significant bit (MSB) of the current remainder, and eventually subtract (XOR) the generator polynomial from the remainder if the MSB had been one, which results in the MSB being cleared. Then shift the remainder left by one bit position, leaving the LSB zero. Finally, feed (XOR) the next payload bit into the LSB position.

In the carry-less calculations the feeded payload bit has no effect unless it reaches the MSB position of the remainder, and is used there for the decision about the eventual XORing of the generator polynomial. Until then it is just shifted left for one place in each step. Therefore, we can delay XORing the payload bit until its position has reached the MSB position. Furthermore, when the last payload bit has been processed in that position, the trailing padded zero-bits can be ignored, respectively, XORed alltogether into the remainder without having any effect on its value.
The instructions for division are now:

XOR the next payload bit with the most significant bit of the current remainder, and if the result is one, then subtract (XOR) the generator polynomial from the remainder. Shift the remainder left by one bit position, leaving the LSB zero.

The CRC calculation now is somewhat shorter:

```
 10100111010    mod 100101

 11111               initial
 111110              shift  (0) <---
+1                   payload
 ------
 011110
  111100             shift
 +0                  payload
  ------
  111100
 -100101             poly
  ------
   11001
   110010            shift
  +1                 payload
   ------
   010010
    100100           shift
   +0                payload
    ------
    100100
   -100101           poly
    ------
     00001
     000010          shift  (4) <---
    +0               payload
     ------
     000010
      000100         shift
     +1              payload
      ------
      100100
     -100101         poly
      ------
       00001
       000010        shift
      +1             payload
       ------
       100010
      -100101        poly
       ------
        00111
        001110       shift
       +1            payload
        ------
        101110
       -100101       poly
        ------
         01011
         010110      shift
        +0           payload
         ------
         010110
          101100     shift
         +1          payload
          ------
          001100
           011000    shift
          +000000    payload+trailer
           ------
           011000
            11000    remainder
            -----
        inv: 00111   CRC
```

### B. Size of Remainder

In this calculation the current remainder seems to be of size $g + 1$ bits. However, we can separate the MSB-and-payload processing from the remainder. The result of that operation is not retained anyway:
As the MSB of the generator polynomial is always equal

to one, subtracting that from $MSB_{remainder} \oplus payload$ is always zero if the subtraction is executed, and is already zero otherwise. Thus, we only need to take care of the rightmost $g$ bits of the remainder, in addition to the bit that pops out on the left in the shift operation.

### C. Table-Lookup

Next we consider using a lookup-table for compressing several steps of the calculation, using a lookup of pre-calculated values. In the above example, to alter the current remainder from step (0) to (4) we need to perform the combination of all the subtractions of the generator polynomial in that area (marked in green), which can be expressed as a single operand. Lets first append (virtually) to the initial remainder all the zero-bits that are caused by the shift operations, and the non-subtractions of the generator polynom.

```
1111100000
-100101             poly
 -000000            no poly
  -100101           poly
   -000000          no poly
----------
1010000010
     -----
```

or, looking only at the remainder's value, we can do

```
 11110  (0)  initial remainder
^11100       some magic value
 -----
 00010  (4)  final reminder
```

It is obvious that in those $n = 4$ steps there are $2^n = 16$ possibilities to subtract or not subtract the generator polynomial at its respective fixed offset, thus, a table of that many entries suffices to calculate the transition from any given remainder value to the remainder value 4 steps ahead, by XORing the appropriate pre-calculated table value.
What is left is to determine which of those table entries to use. Obviously, this depends on the starting value of the remainder and on the four payload bits that are processed in those steps. We consider now all the operations modifying the remainder value that have been left out in the above table approach, in particular, the XOR of the remainder's MSB with the next payload bit (marked in red):

```
 111110
+1                  payload
 +0                 payload
  +1                payload
   +0               payload
 ------
 0101110
```

These top-most four bits are used as the index into our lookup-table: The top-most four bits of the remainder before the initial shift operation, XORed with the next four bits of the payload.
Once we have calculated the new remainder we can repeat the process on the next four bits of the input, until the end of input is reached.
Of course, any other number of steps can be compressed into an appropriate table lookup. In particular, eight bits are frequently used, requiring a lookup-table of 256 entries. The number of bits in the payload should be an integral multiple of the portion size, or otherwise the last (or the first) portion needs to be handled bit-wise or with lookups in a separately designed table lookup.

### D. Constructing the Lookup-Table

Since the values in the lookup-table are constant, they can be pre-calculated without any timing pressure using the slow bit-wise algorithm. For any possible payload of the portion size the difference (in modulo-2 arithmetic) to the final remainder is recorded in the appropriate table entry. Obviously, the table only considers the pure division, so the inversion of initial payload bits and the CRC must be disabled here.

### E. In-Byte Operation

As a consequence of the linear, distributive property of the CRC division we can calculate the CRC of a byte consisting of two nibbles $AB$ as

$$crc(A0 \wedge 0B) = crc(A0) \wedge crc(0B)$$

The terms on the right can be looked up in two tables of 16 entries. So, instead of a 256 byte sized table for byte-mode operation, we only need two tables with a total extension of 32 bytes, of course at the cost of two table lookup operations per payload byte, plus a XOR and some nibble operations. We are still reading the payload in natural byte-wise mode.

### F. Unrolled Loop

In the case of CRC5-checking in USB the payload is fixed at 16 bits, so the loop running thru the two payload bytes can be unrolled into explicit invocations.

## IV. Code Implementations

For better explanation we first present a nibble-mode implementation in C++.

Listing 1. C++ code

```
const unsigned char UsbCrc::crc5Table4[] =
{
        0x00, 0x0E, 0x1C, 0x12, 0x11, 0x1F, 0x0D, 0x03,
        0x0B, 0x05, 0x17, 0x19, 0x1A, 0x14, 0x06, 0x08
};

const unsigned char UsbCrc::crc5Table0[] =
{
        0x00, 0x16, 0x05, 0x13, 0x0A, 0x1C, 0x0F, 0x19,
        0x14, 0x02, 0x11, 0x07, 0x1E, 0x08, 0x1B, 0x0D
};

//---------------
bool
UsbCrc::crc5Check( const unsigned char* data )
//---------------
{
  unsigned char b = data[0] ^ 0x1F;
  unsigned char crc = crc5Table4[b & 0x0F]
              ^ crc5Table0[(b >> 4) & 0x0F];
  b = data[1] ^ crc;
  return (crc5Table4[b & 0x0F]
       ^ crc5Table0[(b>>4) & 0x0F]) == 0x06;
}     //  crc5Check
```

Likewise, an optimized code fragment for Atmel's AVR® microcontroller family ([5], via avr-gcc):

Listing 2. Atmel AVR Assembler code

```
    ; check the CRC5
    ; footprint: 56+32 bytes, 38 clock cycles
    ; - - - - - - - - - - - - - - - - - - -
    ; both these tables have to be located in the
    ; same 256-byte segment !
.section    .progmem.data.crc5Table4,"a",@progbits
    .type   crc5Table4, @object
    .size   crc5Table4, 16
crc5Table4:
    .byte   0x00, 0x0E, 0x1C, 0x12, 0x11, 0x1F, 0x0D, 0x03
    .byte   0x0B, 0x05, 0x17, 0x19, 0x1A, 0x14, 0x06, 0x08
.section    .progmem.data.crc5Table0,"a",@progbits
    .type   crc5Table0, @object
    .size   crc5Table0, 16
crc5Table0:
    .byte   0x00, 0x16, 0x05, 0x13, 0x0A, 0x1C, 0x0F, 0x19
    .byte   0x14, 0x02, 0x11, 0x07, 0x1E, 0x08, 0x1B, 0x0D

.text
    ; check the CRC5
    ; assume Y points to received message, with
    ; payload/crc5 in Y+1 and Y+2
    ; uses 2 registers: rxByte, crc5 for calculation
    ; uses Z registers for progmem pointer
    ldi     ZH, hi8(crc5Table4)     ;   1 clock cycle
    ldd     rxByte, Y+1             ;   2 clock cycles
    ; xor with the initial CRC value of 0x1F
    ldi     crc5, 0x1F              ;   1 clock cycle
    eor     rxByte, crc5            ;   1 clock cycle
    mov     ZL, rxByte              ;   1 clock cycle
    andi    ZL, 0x0F                ;   1 clock cycle
    subi    ZL, lo8(-(crc5Table4))  ;   1 clock cycle
    lpm     crc5, Z                 ;   3 clock cycles
    mov     ZL, rxByte              ;   1 clock cycle
    swap    ZL                      ;   1 clock cycle
    andi    ZL, 0x0F                ;   1 clock cycle
    subi    ZL, lo8(-(crc5Table0))  ;   1 clock cycle
    lpm     rxByte, Z               ;   3 clock cycles
    eor     crc5, rxByte            ;   1 clock cycle
    ; now for the second byte
    ldd     rxByte, Y+2             ;   2 clock cycles
    ; xor with the current CRC5 value
    eor     rxByte, crc5            ;   1 clock cycle
    mov     ZL, rxByte              ;   1 clock cycle
    andi    ZL, 0x0F                ;   1 clock cycle
    subi    ZL, lo8(-(crc5Table4))  ;   1 clock cycle
    lpm     crc5, Z                 ;   3 clock cycles
    mov     ZL, rxByte              ;   1 clock cycle
    swap    ZL                      ;   1 clock cycle
    andi    ZL, 0x0F                ;   1 clock cycle
    subi    ZL, lo8(-(crc5Table0))  ;   1 clock cycle
    lpm     rxByte, Z               ;   3 clock cycles
    eor     crc5, rxByte            ;   1 clock cycle
    ; check for valid residual: reverse of 0x0C = 0x06
    cpi     crc5, 0x06              ;   1 clock cycle
    brne    ignore                  ;   1 clock cycle (nobranch)
```

## V. Resource Usage

As already stated, several other implementations of the CRC5-Checker can be considered, with separate goals.

The byte-mode implementation performs a lookup in a table of 256 bytes for each payload byte. While the CPU clock-cycle consumption of this strategy is only 42% of the nibble-mode's utilization, this comes at the cost of 314% memory usage, possibly further increased by alignment offcuts on the large table.

The bit-mode implementation avoids the memory overhead of a lookup-table, so its memory utilization is only 45% of the nibble-mode's usage, but at the cost of 279% the CPU clock-cycle consumption.

Which implementation to choose depends on the weighting of the restrictions to consider. If both, the memory and the clock cycles, are equally of concern, the solution with the least distance from the origin (null-algorithm) might be the most appropriate, hence, the nibble-mode implementation.

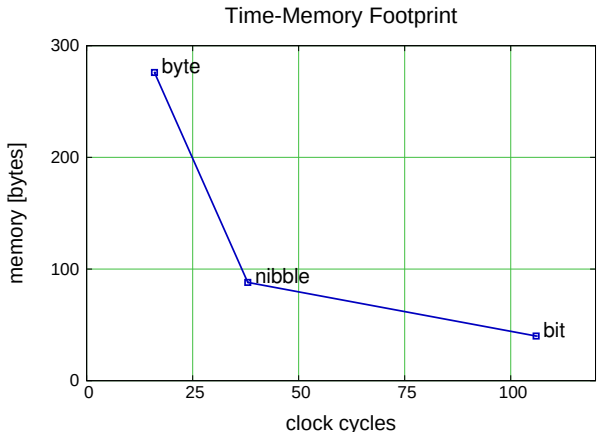


Fig. 2. Memory/Time Tradeoffs for CRC5 Checker Implementations

All resource figures are based on implementations for the AVR® microcontroller architecture with optimized assembly code inlined similar to the code implementation given in listing 2. For other architectures the footprint/performance results might be different, of course.

## VI. Conclusion

We proposed an implementation for checking the validity of CRC5-secured message fields, as used in the USB protocol. An implementation based on 4-bit-wide lookup tables provides an optimal trade-off between memory usage and CPU utilization, compared to other implementation strategies.
While we focus in this paper on the needs of the USB protocol for excatly two bytes long messages (payload+CRC), the algorithm can be easily extended to longer messages.

## References

[1] W. W. Peterson, D. T. Brown
*Cyclic Codes for Error Detection*
Proceedings of the IRE, Vol. 49 pg. 228, 1961

[2] *Universal Serial Bus Specification, Rev. 2.0*
http://www.usb.org

[3] D.V. Sarwate:
*Computation of Cyclic Redundancy Checks via Table-Lookup*
Comm. ACM, vol. 31, no. 8, pp. 1008-1013, Aug. 1988

[4] M. E. Kounavis, F. L. Berry:
*A Systematic Approach to Building High Performance, Software-based, CRC Generators*
Proc. 10th IEEE Symp. Computers and Comm. (ISCC '05), pp. 855-862, June 2005

[5] Atmel®:
*8-bit AVR® Instruction Set*
http://www.atmel.com/images/doc0856.pdf

[6] Ross N. Williams:
*A Painless Guide To CRC Error Detection Algorithms*
v3, Aug 1993
http://www.ross.net/crc/download/crc_v3.txt