Understanding and implementing CRC (Cyclic Redundancy Check) calculation

Table of Contents

- 1. Foreword & Outline
- 2. Introduction to CRC
  - 2.1 CRC verification
- 3. Concept of the CRC shift register
- 4. Implementing CRC-8 algorithms
  - 4.1 Simple CRC-8 shift register implementation for one byte input data
  - 4.2 Modified CRC-8 bitwise implementation for one byte input data
  - 4.3 General CRC-8 bitwise implementation
  - 4.4 Improved CRC-8 byte-by-byte algorithm (lookup table based)
- 5. Extending to CRC-16
- 6. Extending to CRC-32
- 7. CRC algorithm specification
  - 7.1 CRC parametrization
  - 7.2 Representation of generator polynomials
    - 7.2.1 Choosing a generator polynomial
  - 7.3 Reciprocal polynomials
    - 7.3.1 Reversed CRC lookup table and calculation of reciprocal CRC
  - 7.4 Extending to arbitrary CRC sizes (width of polynomials)
- 8. Additional remarks (points worth to know)
  - 8.1 Basic mathematical view of CRC
  - 8.2 Background to CRC verification
  - 8.3 CRC-1 is the same as a parity bit
  - 8.4 Why is addition is the same as subtraction in CRC arithmetic?
  - 8.5 Why does multiplication with x^n append n zeros?
  - 8.6 When using an initial value other than zero in the shift register, the result is incorrect.
- 9. Conclusion & References

View Online CRC Silverlight application now!
View Online CRC Javascript application now!
Download C# source code (37 kb)

Article updated March 2023 (for details, click here to see the changelog).

1. Foreword & Outline

[Back to top]

This article is the result of the fact that I found finally time to deal with CRC. After reading Wikipedia and some other articles, I had the feeling to not really understand *completely in depth*.
Therefore I decided to write this article, trying to cover all topics I had difficulties with. And this in exactly the same order I concerned myself with CRC. Please note that this article is not indented to be a full comprehensive CRC guide explaining all details - it should be used as an additional, practical oriented note to all general explanations on the web.

Here's the outline:

- At first, the general idea and functionality of CRC is discussed.
- Subsequently, some examples are calculated by hand to get familiar with the process of CRC calculation.

- Based on those observations, implementations of CRC calculation are presented step by step, from naive ones till more efficient algorithms. Here the emphasis lies on the target to really get the point of the code compared to the manual calculation. Here an exemplary CRC-8 polynomial is used.
- Afterwards, the achieved knowledge is expanded to CRC-16 and CRC-32 calculation, followed by some CRC theory and maybe a FAQ section.

[Back to top]

2. Introduction

CRC (*Cyclic Redundancy Check*) is a checksum algorithm to detect inconsistency of data, e.g. bit errors during data transmission. A checksum, calculated by CRC, is attached to the data to help the receiver to detect such errors. Refer also to [1] for a short or to [4] for a very detailed CRC introduction.

CRC is based on division. The actual input data is interpreted as one long binary bit stream (dividend) which is divided by another fixed binary number (divisor). The remainder of this division is the checksum value.
However, reality is a bit more complicated. The binary numbers (dividend and divisor) are not treated as normal integer values, but as binary polyonimals where the actual bits are used as coefficients.

For example, the input data 0x25 = 0010 0101 is taken as $0*x^7 + 0*x^6 + 1*x^5 + 0*x^4 + 0*x^3 + 1*x^2 + 0*x^1 + 1*x^0$.

Division of polynomials differs from integer division. Without going into detail, the underlying used aritmetic for CRC calculation is based on the XOR (Exclusive-OR) operation (we'll come to an example soon!).
- The dividend is the complete input data (interpreted as binary stream).
- The divisor, also called *generator polynomial*, is statically defined by the used CRC algorithm. CRC-*n* using a fixed defined generator polynom with (n+1) bits.
- The CRC checksum value is defined as dividend % divisor.

XOR	0	1
0	0	1
1	1	0

XOR truth table

For manual calculation, *n* zero bits are appended to the input data before actual CRC calculation (polynomial division) is computed. Let's perform an example CRC computation:

Example:

Input data is the byte 0xC2 = b11000010.
As generator polynomial (=divisor), let's use b100011101.
The divisor has 9 bits (therefore this is a CRC-8 polynomial), so append 8 zero bits to the input pattern .
Align the leading '1' of the divisor with the first '1' of the dividend and perform a step-by-step school-like division, using XOR operation for each bit:

```
ABCDEFGHIJKLMNOP
110000100000000
100011101
---------
010011001
 100011101
 ----------
 000101111
    100011101          (*)
    ---------
    001100101
```

```
      100011101
      ---------
      010001001
       100011101
       ---------
       000001111 = 0x0F
ABCDEFGHIJKLMNOP
```

The actual CRC value is 0x0F.

Useful observations:

- In each step, the leading '1' of the divisor is always aligned with the first '1' of the dividend. This implies that the divisor does not move only 1 bit right per step, but sometimes also several steps (e.g. like in line (*).
- The algorithms stops if the divisor zeroed out each bit of the actual input data (without padding bytes): The input data ranges from column A to H including. In the last step, column H and all prior columns contain 0, so the algorithm stops.
- The remainder (= CRC) is the value 'below' the padding zero bits (column I to P). Because we added n padding bytes, the actual CRC value has also n bits.
- Only the remainder in each step is of interest, the actual division result (quotient) is therefore not tracked at all.

2.1 CRC Verification

[Back to top]

The remainder is the CRC value which is transmitted along with the input data to the receiver. The receiver can either verify the received data by computing the CRC and compare the calculated CRC value with the received one. Or, more commonly used, the CRC value is directly appened to the actual data. Then the receiver computes the CRC over the whole data (input with CRC value appended): If the CRC value is 0, then most likely no bit error occured during transmission. Let's do verification according the latter case:

Example verification:

The actual transmission data (input data + CRC) would be b1100001000001111 Note that we have used an 8bit CRC, so the actual CRC value is also 8bit long. The generator polynomial is statically defined by the used CRC algorithm and so it's known by the receiver.

```
ABCDEFGHIJKLMNOP
1100001000001111
100011101.......
---------.......
010011001.......
 100011101.....
 ---------.....
 000101111.....
    100011101..
    ---------...
    001100100...
      100011101
      ---------.
      010001110.
       100011101
       ---------
```

```
       000000000 -> Remainder is 0, data ok!
ABCDEFGHIJKLMNOP
```

3. Concept of the CRC shift register

[Back to top]

So we have seen how to calculate the CRC checksum value manually, but how can it be implemented?
The input data stream is generally quite long (more than 1 bit) so it's not possible to perform a simple division like "Input data % generator polynomial". The computation has to be performed step-by-step and here the concept of a shift register comes into play.

A shift register has a fixed width and can shift it's content by one bit, removing the bit at the right or left border and shift in a new bit at the freed position. CRC uses a left shift register: When shifted, the most significant bit pops out the register, the bit at position MSB-1 moves one position left to postion MSB, the bit at position MSB-2 to MSB-1 and so on. The bit position of the least significant bit is free: here the next bit of the input stream is shifted in.

```
       MSB                  LSB
       --- --- --- --   -- ---
  <-- |   |   |   |....    |   | <-- (shift in input message bits)
       --- --- --- --   -- ---
```

The process of CRC calculation using a shift register is as follow:

1. Initialize the register with 0. (Note that initial values != 0 are handled especially, consider chapter 8.6).
2. Shift in the input stream bit by bit. If the popped out MSB is a '1', XOR the register value with the generator polynomial.
3. If all input bits are handled, the CRC shift register contains the CRC value.

Let's visualize the procedure with the example data from above

CRC-8 Shift Register Example: Input data = 0xC2 = b11000010 (with 8 zero bits appended: b1100001000000000), Polynomial = b100011101

```
1. CRC-8 register initialized with 0.
         --- --- --- --- --- --- --- ---
        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  <-- b1100001000000000
         --- --- --- --- --- --- --- ---

2. Left-Shift register by one position. MSB is 0, so nothing do happen, shift in next byte of
input stream.
         --- --- --- --- --- --- --- ---
        | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |  <-- b1000010000000000
         --- --- --- --- --- --- --- ---

3. Repeat those steps. All steps are left out until there is a 1 in the MSB (nothing
interesting happens), then the state looks like:
         --- --- --- --- --- --- --- ---
        | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |  <-- b00000000
         --- --- --- --- --- --- --- ---

4. Left-Shift register. MSB 1 pops out:
         --- --- --- --- --- --- --- ---
   1 <- | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |  <-- b0000000
         --- --- --- --- --- --- --- ---
  So XOR the CRC register (with popped out MSB) b110000100 with polynomial b100011101 =
b010011001 = 0x99. The MSB is discarded, so the new CRC register value is 010011001:
```

```
         --- --- --- --- --- --- --- ---
        | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |  <-- b0000000
         --- --- --- --- --- --- --- ---

5. Left-Shift register. MSB 1 pops out: b100110010 ^ b100011101 = b000101111 = 0x2F:
         --- --- --- --- --- --- --- ---
        | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |  <-- b000000
         --- --- --- --- --- --- --- ---

6. Left-shift register until a 1 is in the MSB position:
         --- --- --- --- --- --- --- ---
        | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |   <-- b0000
         --- --- --- --- --- --- --- ---

7. Left-Shift register. MSB 1 pops out: b101111000 ^ b100011101 = b001100101 = 0x65:
         --- --- --- --- --- --- --- ---
        | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |   <-- b000
         --- --- --- --- --- --- --- ---

8. Left-shift register until a 1 is in the MSB position:
         --- --- --- --- --- --- --- ---
        | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |   <-- b00
         --- --- --- --- --- --- --- ---

9. Left-Shift register. MSB 1 pops out: b110010100 ^ b100011101 = b010001001 = 0x89:
         --- --- --- --- --- --- --- ---
        | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |   <-- b0
         --- --- --- --- --- --- --- ---

10. Left-Shift register. MSB 1 pops out: b10001001 ^ b100011101 = b000001111 = 0x0F:
         --- --- --- --- --- --- --- ---
        | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |   <-- <empty>
         --- --- --- --- --- --- --- ---

All input bits are processed, the algorithm stops. The shift register contains now the CRC
value which is 0x0F.
```

4. Implementing CRC-8 algorithms

[Back to top]

This chapter handles different algorithms and their implementations in C# for calculating CRC-8 checksum values. It starts with simple algorithms for limited input data and ends with efficient table-based implementations.

4.1 Simple CRC-8 shift register implementation for one byte input data

[Back to top]

Let's start with an implementation of a CRC-8 algorithm for solely one byte input data. The implementation will stay very closely to the shift register process from the example above.

A CRC-8 algorithm uses actually a 9bit generator polynomial, but it would be cumbersome to track such an unaligned value in an algorithm. Fortunately, as described in the previous chapter, the most significant bit can be discarded. First, it is always 1. Second, because the divisor is always aligned in such a manner that this leading '1' alignes with the next '1' of the dividend, the XOR result for this bit is always 0.
This means we leave out the MSB '1', so we can use the generator polynomial b~~1~~00011101 = 0x1D as the example polynomial from now on.

Let's start with an implementation which is as close as possible to the shift register approach:

```
public static byte Compute_CRC8_Simple_OneByte_ShiftReg(byte byteVal)
{
    const byte generator = 0x1D;
    byte crc = 0; /* init crc register with 0 */
    /* append 8 zero bits to the input byte */
    byte[] inputstream = new byte[] { byteVal, 0x00 };
    /* handle each bit of input stream by iterating over each bit of each input byte */
    foreach (byte b in inputstream)
    {
        for (int i = 7; i >= 0; i--)
        {
            /* check if MSB is set */
            if ((crc & 0x80) != 0)
            {   /* MSB set, shift it out of the register */
                crc = (byte)(crc << 1);
                /* shift in next bit of input stream:
                 * If it's 1, set LSB of crc to 1.
                 * If it's 0, set LSB of crc to 0. */
                crc = ((byte)(b & (1 << i)) != 0) ? (byte)(crc | 0x01) : (byte)(crc & 0xFE);
                /* Perform the 'division' by XORing the crc register with the generator
polynomial */
                crc = (byte)(crc ^ generator);
            }
            else
            {   /* MSB not set, shift it out and shift in next bit of input stream. Same as
above, just no division */
                crc = (byte)(crc << 1);
                crc = ((byte)(b & (1 << i)) != 0) ? (byte)(crc | 0x01) : (byte)(crc & 0xFE);
            }
        }
    }
    return crc;
}
```

4.2 Modified CRC-8 bitwise implementation for one byte input data

[Back to top]

Well, above implementation looks complicated! How can it be simplified?

- The first 8 left-shifts are useless because the CRC value is initialized with 0 so no XOR operation is performed. This means we can initialize the CRC value directly with the input byte.
- So only '0' are left in the input stream (the appended zeros). They do not have to be explcitely shifted-in, as the C# leftshift operator << fills in the LSB with '0' by default.
- This implies that the inputstream array is not required anymore.

Applying these simplifications result in following implementation (much better, isn't it?):

```
public static byte Compute_CRC8_Simple_OneByte(byte byteVal)
{
    const byte generator = 0x1D;
    byte crc = byteVal; /* init crc directly with input byte instead of 0, avoid useless 8
bitshifts until input byte is in crc register */

    for (int i = 0; i < 8; i++)
    {
        if ((crc & 0x80) != 0)
        { /* most significant bit set, shift crc register and perform XOR operation, taking not-
saved 9th set bit into account */
            crc = (byte)((crc << 1) ^ generator);
        }
        else
        { /* most significant bit not set, go to next bit */
            crc <<= 1;
        }
    }
```

```
    return crc;
}
```

To illustrate how the algorithmus is working, the example from above (input byte 0xC2, generator polynomial 0x1D) is repeated - this time showing the intermediate values of each step of implementation *Compute_CRC8_Simple_OneByte*.

Step-by-step visualization of simple CRC-8 algorithmus:

```
1100001000000000            crc = 0xC2
 100011101                  i = 0: 0xC2 = b11000010 -> MSB set:
 ---------
 010011001                  i = 0: crc = (crc << 1) ^ generator = (0xC2 << 1) ^ 0x1D = 0x184 ^
                            0x1D = 0x99.
  100011101                 i = 1: 0x99 = b10011001 -> MSB set:
  ---------
  000101111                 i = 1: crc = (0x99 << 1) ^ 0x1D = 0x132 ^ 0x1D = 0x2F
   100011101                i = 2: 0x2F = b00101111 -> MSB not set: crc = (0x2F << 1) = 0x5E
    100011101               i = 3: 0x5E = b01011110 -> MSB not set: crc = (0x5E << 1) = 0xBC
     100011101              i = 4: 0xBC = b10111100 -> MSB set:
     ---------
     001100101              i = 4: crc = (0xBC << 1) ^ 0x1D = 0x178 ^ 0x1D = 0x65
      100011101             i = 5: 0x65 = b01100101 -> MSB not set: crc = (0x65 << 1) = 0xCA
       100011101            i = 6: 0xCA = b11001010 -> MSB set:
       ---------
       010001001            i = 6: crc = (0xCA << 1) ^ 0x1D = 0x194 ^ 0x1D = 0x89
        100011101           i = 7: 0x89 = b10001001 -> MSB set:
        ---------
        000001111           i = 7: crc = (0x89 << 1) ^ 0x1D = 0x112 ^ 0x1D = 0x0F
```

The fact that the crc value is left-shifted by one _before_ it's 'xored' with the divisor should become clear: it's due to the already discussed reason that the MSB bit of the generator polynomial is not stored / not used by the algorithm as well as it's result.

4.3 General CRC-8 bitwise implementation

[Back to top]

Till now only one byte was used as input data, so let's see what happens if the input data is extended to a byte array.
The first function *Compute_CRC8_Simple_OneByte_ShiftReg()* could easily be adapted (only the input parameter would be a byte array at which a 0x00 byte is appended), but what about *Compute_CRC8_Simple_OneByte()*?
The interesting point is at the border between two bytes: If one byte is completely processed, how is the subsequent byte incorporated into the calculation process? Again, let's start with a simple example (even more manual CRC-calculation action!):

Example for two byte input data {0x01, 0x02} with polynomial 0x1D

```
0000000100000010 00000000
       100011101
       ---------
       000011111 0000
           100011101
           ---------
```

```
          0111011010
           100011101
           ---------
           0110001110
            100011101
            ---------
            0100100110
             100011101
             ---------
             0001110110 = 0x76
```

Actually the algorithm handles one byte at a time, and does not consider the next byte until the current one is completely processed. Refering to *Compute_CRC8_Simple_OneByte*, the value *crc* is set to 0x01 and the input looks like 00000001**00000000**... So let's see the state when the first byte is completely processed:

```
00000001 0000000000000000
       100011101
       ---------
       000011101
```

Compare this to our manual approach where we have the second byte 0x02 already 'in range':

```
00000001 00000010 00000000
       100011101
       ---------
       000011111
```

So obviously the next byte has to be XORed with the current CRC value: 000011101 ^ 00000010 = 000011111 and the algorithm then continues with the 'new' *xored* value.

With this knowledge we can easily extend our algorithm to work with an input byte array of arbitrary length:

```
public static byte Compute_CRC8_Simple(byte[] bytes)
{
    const byte generator = 0x1D;
    byte crc = 0; /* start with 0 so first byte can be 'xored' in */

    foreach (byte currByte in bytes)
    {
        crc ^= currByte; /* XOR-in the next input byte */

        for (int i = 0; i < 8; i++)
        {
            if ((crc & 0x80) != 0)
            {
                crc = (byte)((crc << 1) ^ generator);
            }
            else
            {
                crc <<= 1;
            }
        }
    }

    return crc;
}
```

4.4 Improved CRC-8 byte-by-byte algorithm (lookup table based)

[Back to top]

So far the algorithm is quite inefficient as it works bit by bit. For larger input data, this could be quite slow. But how can our CRC-8 algorithm be accelerated?

The dividend is the current crc byte value - and a byte can only take 256 different values. The polynomial (= divisor) is fixed. Why not precompute the division for each possible byte by the fixed polynomial and store these result in a lookup table? This is possible

as the remainder is always the same for the same dividend and divisor! Then the input stream can be processed byte by byte instead of bit by bit.

Let's use our common example to demonstrate the process manually:

Process for bytewise CRC-8 using input data {0x01, 0x02} and polynomial 0x1D

1. Init crc = 0x00.
2. *'Xor-in'* next input byte 0x01: 0x00 ^ 0x01 = 0x01.
3. Calculate CRC-8 of 0x01 using precomputed lookup table: table[0x01] = crc = 0x1D.
4. *'Xor-in'* next input byte 0x02: 0x1D ^ 0x02 = 0x1F.
5. Calculate CRC-8 of 0x1F: using precomputed lookup table: table[0x1F] = crc = 0x76.

For step 3 and 5, the lookup table is used instead of bit by bit processing - that makes the actual speedup.

Here an implementation for calculating the 256-element lookup table:

```
public static void CalculateTable_CRC8()
{
    const byte generator = 0x1D;
    crctable = new byte[256];
    /* iterate over all byte values 0 - 255 */
    for (int dividend = 0; dividend < 256; dividend++)
    {
        byte currByte = (byte)dividend;
        /* calculate the CRC-8 value for current byte */
        for (byte bit = 0; bit < 8; bit++)
        {
            if ((currByte & 0x80) != 0)
            {
                currByte <<= 1;
                currByte ^= generator;
            }
            else
            {
                currByte <<= 1;
            }
        }
        /* store CRC value in lookup table */
        crctable[dividend] = currByte;
    }
}
```

Then improved CRC-8 algorithm using the lookup table is as follow:

```
public static byte Compute_CRC8(byte[] bytes)
{
    byte crc = 0;
    foreach (byte b in bytes)
    {
        /* XOR-in next input byte */
        byte data = (byte)(b ^ crc);
        /* get current CRC value = remainder */
        crc = (byte)(crctable[data]);
    }

    return crc;
}
```

The improvement of speed comes at the cost of processing time to precalculate the table and of higher memory consumption of the 256-byte elements, but that's worth it :-)!

5. Extending to CRC-16

[Back to top]

The more bits the CRC value has the less is the probability of a collision: for CRC-8 there are only 256 different CRC values. This means if the data is disturbed or modified between sender and receiver, there is a probability of 1/256 that the modified data stream

has the same CRC value as the original data stream, thus the error is not detected. Beside other factors (more on this in the theory part), increasing the CRC width results in better error protection.

This raises the question: What is the impact on the implementation if we want to extend it from CRC-8 to CRC-16?

- 1. Obviously a CRC-16 uses a polynomial of degree 16 with 17 terms, but similar to CRC-8 the most significant bit is implicitly 1. Therefore the generator polynomial as well as the CRC value have now a 16bit data type.
- 2. How to 'Xor-In' the next input byte (8bit) into the CRC value (16bit)? The answer is into the most significant byte of the CRC. Similar to chapter 4.3, let's visualize it with an example:

First, let's make another manual calculation, this time for CRC-16 with polynomial 0x1021:

Example for two byte input data {0x01, 0x02} with polynomial 0x1021 (1 0001 0000 0010 0001)

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0000000100000010000000000000000
       10001000000100001
       -----------------
       00001001000100001 = 0x1221 (intermediate CRC value after completing first byte)
           10001000000100001
           -----------------
           00011001000110001
              10001000000100001
              -----------------
              01000000110101001
               10001000000100001
               -----------------
               00001001101110011 = 0x1373 (final CRC value after both bytes)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

What happens if the algorithm has handled the first input byte 0x01:

```
0000000100000000000000000000000
       10001000000100001
       -----------------
       00001000000100001
```

Here we see that the next input byte 0x02 = 00000010 has be xored into the MSB of 00001000000100001 to get 00001001000100001 to proceed.

- 3. The check if the most significant bit set changes because bit15 instead of bit7 has to be tested: 0x80 -> 0x8000

Therefore the CRC-16 simple implementation looks like:

```
public static ushort Compute_CRC16_Simple(byte[] bytes)
{
    const ushort generator = 0x1021; /* divisor is 16bit */
    ushort crc = 0; /* CRC value is 16bit */

    foreach (byte b in bytes)
    {
        crc ^= (ushort(b << 8); /* move byte into MSB of 16bit CRC */

        for (int i = 0; i < 8; i++)
        {
            if ((crc & 0x8000) != 0) /* test for MSB = bit 15 */
            {
                crc = (ushort((crc << 1) ^ generator);
            }
            else
```

```
            {
                crc <<= 1;
            }
        }
    }

    return crc;
}
```

The modification of the implementations for calculating the CRC-16 lookup table is now quite easy:

```
public static void CalculateTable_CRC16()
{
    const ushort generator = 0x1021;
    crctable16 = new ushort[256];

    for (int dividend = 0; dividend < 256; dividend++) /* iterate over all possible input byte values 0 - 255 */
    {
        ushort curByte = (ushort(dividend << 8); /* move dividend byte into MSB of 16Bit CRC */

        for (byte bit = 0; bit < 8; bit++)
        {
            if ((curByte & 0x8000) != 0)
            {
                curByte <<= 1;
                curByte ^= generator;
            }
            else
            {
                curByte <<= 1;
            }
        }

        crctable16[dividend] = curByte;
    }
}
```

The actual byte by byte is a bit tricky so let's first check our example again:

Process for bytewise CRC-16 using input data {0x01, 0x02} and polynomial 0x1021

```
1. Init crc = 0
2. Handle first input byte 0x01:
 2.1 'Xor-in' first input byte 0x01 into MSB(!) of crc:
   0000 0000 0000 0000 (crc)
   0000 0001 0000 0000 (input byte 0x01 left-shifted by 8)
   -------------------
   0000 0001 0000 0000 = 0x0100
   The MSB of this result is our current dividend: MSB(0x100) = 0x01.
 2.2 So 0x01 is the dividend. Get the remainder for dividend from our table: crctable16[0x01] = 0x1021. (Well this value is famila from the manual computation above.)
   Remember the current crc value is 0x0000. Shift out the MSB of current crc and xor it with the current remainder to get the new CRC:
   0001 0000 0010 0001 (0x1021)
   0000 0000 0000 0000 (CRC 0x0000 left-shifted by 8 = 0x0000)
   -------------------
   0001 0000 0010 0001 = 0x1021 = intermediate crc.
3. Handle next input byte 0x02:
 Currently we have intermediate crc = 0x1021 = 0001 0000 0010 0001.
 3.1 'Xor-in' input byte 0x02 into MSB(!) of crc:
   0001 0000 0010 0001 (crc 0x1021)
   0000 0010 0000 0000 (input byte 0x02 left-shifted by 8)
   -------------------
   0001 0010 0010 0001 = 0x1221
```

```
   The MSB of this result is our current dividend: MSB(0x1221) = 0x12.
 3.2 So 0x12 is the dividend. Get the remainder for dividend from our table: crctable16[0x12] =
0x3273.
   Remember the current crc value is 0x1021. Shift out the MSB of current crc and xor it with
the current remainder to get the new CRC:
   0011 0010 0111 0011 (0x3273)
   0010 0001 0000 0000 (CRC 0x1021 left-shifted by 8 = 0x2100)
   -------------------
   0001 0011 0111 0011 = 0x1373 = final crc.
```

The important point is here that after xoring the current byte into the MSB of the intermediate CRC, the MSB is the index into the lookup table.

Here the corresponding implementation for the table-based CRC-16 algorithm:

```
public static ushort Compute_CRC16(byte[] bytes)
{
    ushort crc = 0;
    foreach (byte b in bytes)
    {
        /* XOR-in next input byte into MSB of crc, that's our new intermediate dividend */
        byte pos = (byte)( (crc >> 8) ^ b); /* equal: ((crc ^ (b << 8)) >> 8) */
        /* Shift out the MSB used for division per lookuptable and XOR with the remainder */
        crc = (ushort)((crc << 8) ^ (ushort)(crctable16[pos]));
    }
    return crc;
}
```

6. Extending to CRC-32

[Back to top]

After having understood the extension process from CRC-8 to CRC-16 in the previous chapter, the modifications to CRC-32 is pretty easy. I discard the detailed steps as they are very similar to the CRC-16 case - just a quick overview of the changes:

- CRC-32 uses a 33-bit polynom, however again the most signficant bit is always '1' and can be discarded. Therefore, the polynomial and the CRC value are represented by 32 bit variables.
- The bit operations change slightly: Moving the input byte into the MSB of CRC requires now a shift by 24. Also the check for the most significant bit uses a different mask of 0x8000000 instead of 0x8000.

Actually that's it, so here is the bitwise CRC-32 algorithm implementation:

```
public uint Compute_CRC32_Simple(byte[] bytes)
{
    const uint polynomial = 0x04C11DB7; /* divisor is 32bit */
    uint crc = 0; /* CRC value is 32bit */

    foreach (byte b in bytes)
    {
        crc ^= (uint)(b << 24); /* move byte into MSB of 32bit CRC */

        for (int i = 0; i < 8; i++)
        {
            if ((crc & 0x80000000) != 0) /* test for MSB = bit 31 */
            {
                crc = (uint)((crc << 1) ^ polynomial);
            }
            else
            {
                crc <<= 1;
            }
        }
    }

    return crc;
}
```

Calculating the CRC-32 lookup table and using it for the CRC computation is then straight forward:

```
private void CalculateCrcTable_CRC32()
{
    const uint polynomial = 0x04C11DB7;
    crcTable = new uint[256];

    for (int dividend = 0; dividend < 256; dividend++) /* iterate over all possible input byte values 0 - 255 */
    {
        uint curByte = (uint)(dividend << 24); /* move dividend byte into MSB of 32Bit CRC */
        for (byte bit = 0; bit < 8; bit++)
        {
            if ((curByte & 0x80000000) != 0)
            {
                curByte <<= 1;
                curByte ^= polynomial;
            }
            else
            {
                curByte <<= 1;
            }
        }

        crcTable[dividend] = curByte;
    }
}
```

```
public uint Compute_CRC32(byte[] bytes)
{
    uint crc = 0;
    foreach (byte b in bytes)
    {
        /* XOR-in next input byte into MSB of crc and get this MSB, that's our new intermediate dividend */
        byte pos = (byte)((crc ^ (b << 24)) >> 24);
        /* Shift out the MSB used for division per lookuptable and XOR with the remainder */
        crc = (uint)((crc << 8) ^ (uint)(crcTable[pos]));
    }

    return crc;
}
```

7. CRC algorithm specification

[Back to top]

At this point, you (and me hopefully... ;-)) should know how CRC computation works and how to calculate it manually as well as how to implement it in your favorite programming language.

However, a CRC instance is defined by specific parameters. We have already seen that e.g. the width of the generator polynomial varies, but there are more definition parameters. In order to be able to implement each CRC instance, let's discuss how an CRC algorithm instance is described.

7.1 CRC parametrization

[Back to top]

Following standard parameters are used to define a CRC algorithm instance:

- **Name**: Well, a CRC instance has to be identified somehow, so each public defined CRC parameter set has a name like e.g. CRC-16/CCITT.
- **Width** (in bits): Defines the width of the result CRC value (n bits). Simultaneously, also the width of the generator polynomial is defined (n+1 bits). Most common used widths are 8, 16 and 32 bit. But theoretically all widths beginning from 1 are possible. In practice, even quite big (80 bit) or uneven (5 bit or 31 bit) widths are used.
- **Polynomial**: Used generator polynomial value. Note that different representations exist, see chapter 7.2.
- **Initial Value**: The value used to initialize the CRC value / register. In the examples above, always zero is used, but it could be any value.
  CAUTION: See chapter 8.6 for the correct handling of initial values not equal to 0.

- **Input reflected**: If this value is TRUE, each input byte is reflected before being used in the calculation. Reflected means that the bits of the input byte are used in reverse order. So this also means that bit 0 is treated as the most significant bit and bit 7 as least significant.
  Example with byte 0x82 = b10000010: Reflected(0x82) = Reflected(b10000010) = b01000001 = 0x41.
- **Result reflected**: If this value is TRUE, the final CRC value is reflected before being returned. The reflection is done over the whole CRC value, so e.g. a CRC-32 value is reflected over all 32 bits.
- **Final XOR value**: The Final XOR value is xored to the final CRC value before being returned. This is done after the 'Result reflected' step. Obviously a Final XOR value of 0 has no impact.
- **Check value** [Optional]: This value is not required but often specified to help to validate the implementation. This is the CRC value of input string "123456789" or as byte array: [0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39].

For a great overview over standard CRC algorithms, refer to [5].

Here a pseudo-code CRC-32 table-based implementation taking the definition parameters into account:

```
public uint Compute_CRC32(byte[] bytes)
{
    uint crc = crcModel.Initial; /* CRC is set to specified initial value */
    foreach (byte b in bytes)
    {
        /* reflect input byte if specified, otherwise input byte is taken as it is */
        byte curByte = (crcModel.InputReflected ? CrcUtil.Reflect8(b) : b);

        /* XOR-in next input byte into MSB of crc and get this MSB, that's our new intermediate
dividend */
        byte pos = (byte)((crc ^ (curByte << 24)) >> 24);
        /* Shift out the MSB used for division per lookuptable and XOR with the remainder */
        crc = (uint)((crc << 8) ^ (uint)(crcTable[pos]));
    }
    /* reflect result crc if specified, otherwise calculated crc value is taken as it is */
    crc = (crcModel.ResultReflected ? CrcUtil.Reflect32(crc) : crc);
    /* Xor the crc value with specified final XOR value before returning */
    return (uint)(crc ^ crcModel.FinalXor);
}
```

And finally a possible implementation to reflect a 16bit value:

```
public static ushort Reflect16(ushort val)
{
    ushort resVal = 0;

    for (int i = 0; i < 16; i++)
    {
        if ((val & (1 << i)) != 0)
        {
            resVal |= (ushort)(1 << (15 - i));
        }
    }

    return resVal;
}
```

7.2 Representation of generator polynomials

[Back to top]

It is worth to know that there are different ways to represent a generator polynomial in hexadecimal.

Remember: A CRC-n uses a generator polynomial of degree n of the form $x^n + x^{n-1} + ... + x^1 + x^0$. Note that it has n+1 coefficients.

- Normal representation: The most significant bit (= x^n) of the generator polynomial is left out in the hexadecimal representation (as it's always 1). The hexadecimal polynomial contains only the coefficients x^{n-1} ... x^0. We used normal respresentation in this article.
  Example: In the CRC-8 discussion we used polynomial 100011101. Discarding the most significant bit results in 100011101, which is 0x1D.
- Reversed representation: The most significant bit (= x^n) of the generator polynomial is left out in the hexadecimal representation (like in the normal representation), but the tail is then reflected ('LSB first'), i.e. each nibble is reversed. So the most-significant bit does not match x^{n-1} as in the normal representation, but x^0.

Example: Polynomial is 100011101. Discarding the most significant bit results in ~~1~~0001 1101. Reflection (reverse of each nibble) of 0001 1101 is 1011 1000 = 0xB8.

- Koopman representation: The least(!) significant bit (= x^0) of the generator polynomial is discarded. The hexadecimal polynomial contains only the coefficients x^n ... x^1.
  Example: Polynomial is 100011101. Discarding the least significant bit results in 10001110~~1~~ = 0x8E.

Note that there are also so-called reciprocal polynomials which have their own representation, see chapter 7.3.

INFO:

Please note that this only affects the *representation* of the polynomial. But the key point is that in each case the SAME polynomial is used for calculation.

Consider the often used CRC-16 polynomial x^16 + x^12 + x^5 + 1. In binary, that's 1 0001 0000 0010 0001.

Here the three different representations for this example:

```
1 0001 0000 0010 0001
  0001 0000 0010 0001                                  = 0x1021 - Normal Representation
  0001 0000 0010 0001 -> reversed  : 1000 0100 0000 1000 = 0x8408 - Reversed representation
1 0001 0000 0010 000  -> rearranged: 1000 1000 0001 0000 = 0x8810 - Koopman representation
```

7.2.1 Choosing a generator polynomial

[Back to top]

The length on the generator polynomial depends on the maximum length of the input data and the desired error detection properties. The more likely one or more bit errors shall be detected and/or the longer the input data may be, the longer the generator polynomial has to be. This also decreases the probality of collisions (same CRC value for different input data).
Note that the actual value of the generator polynomial has also major impact on its error detection capabilities and its design is non-trivial and requires serious math knowledge (e.g. see [2] for an expert article.)

7.3 Reciprocal polynomials

[Back to top]

Reciprocal polynomials are polynomials that are reflected. So the least significant coefficient becomes the most significant and the other way. In other words, a reciprocal polynomial is created from a polynomial by assigning coefficient x^n to x^0, x^{n-1} to x^1 and so on. The representation is based on the Koopman representation, but reflected ('LSB first').

CAUTION: Sometimes reciprocal polynomials are called reversed polynomials. This can easily be mixed up with the reversed representation of polynomial as described in chapter 7.2. However these are two different things.

Example: Keep with the CRC-16 polynomial 0x1021 which has binary representation 1 0001 0000 0010 0001.
The Koopman representation is 1000 1000 0001 0000 ~~1~~.
Reversing (or reflecting) the polynomial results in 0000 1000 0001 0001 = 0x0811.

It's a fact that reversed polynomials are 'as good as' the polynomials of which they are the reciprocal ones referring to their error detection properties.
Unfortunately such polynomials make it all a bit more complicated: the calculated CRC value of a polynomial is **NOT** the same as the calculated CRC value of its reciprocal polynomial for the same message data!

Because the LSB and MSB are exchanged, you can think of the processing of reversed polynomials as they would be shifted from the other side into the CRC shift register: In the example of the CRC shift register in chapter 3 the input data was shifted from the right. If the reversed polynomial is used, you could get the same CRC result when shifting the input data from the left. Actually the whole processing is then just mirrored.

In the C# download package linked at the top, there is a 'reflected' implementation for each CRC class: Such a reflected CRC algorithm produces the same CRC result when using a reversed polynomial like the standard implementation using the non-reversed polynomial by 'mirroring' the processing.

7.3.1 Reversed CRC lookup table and calculation of reciprocal CRC

[Back to top]

The fact that there are the two variants MSB-to-LSB and LSB-to-MSB often causes confusion. For example this is the reason why often two different lookup tables are found in the net for the same CRC instance: For the well-known standard CRC-32 instance (e.g. PKZIP) with polynomial 0x04c11db7, you can find a lookup table starting with the values 0x00000000 and 0x04C11DB7, the other one with 0x00000000 and 0x77073096. The first one correspond to the algorithms described in this article, while the second corresponds to the reciprocal variant where the coefficients are reflected.
Let's recall the calculation for the 'normal' CRC-32 lookup table, compared to the 'reciprocal variant' of the CRC-32 lookup table :

```
private void CalculateCrcTable()
{
    crcTable = new uint[256];

    for (int dividend = 0; dividend < 256;
dividend++)
    {
        uint curByte = (uint)(dividend << 24);
        for (byte bit = 0; bit < 8; bit++)
        {
            if ((curByte & 0x80000000) != 0)
            {
                curByte <<= 1;
                curByte ^= polynomial;
            }
            else
            {
                curByte <<= 1;
            }
        }

        crcTable[dividend] = curByte;
    }
}
```

```
private void CalculateCrcTableReciprocal()
{
    crcTable = new uint[256];

    for (int dividend = 0; dividend < 256;
dividend++)
    {
        uint curByte = (uint)(dividend);
        for (byte bit = 0; bit < 8; bit++)
        {
            if ((curByte & 0x00000001) != 0)
            {
                curByte >>= 1;
                curByte ^=
Reflect32(polynomial);
            }
            else
            {
                curByte >>= 1;
            }
        }

        crcTable[dividend] = curByte;
    }
}
```

To calculate the reciprocal variant lookup-table, two changes are required to handle the reflected bit-order: At first, the CRC parameter *Polynomial* has to be reflected, and second the order of bit processing in the algorithm itself needs to be changed from MSB-to-LSB to LSB-to-MSB, resulting in right-shifting instead of left-shifting.
So this results in two different look-up tables. This implies that also the actual CRC calculation needs to adapated for the reciprocal variant in order to retrieve the same CRC result as for the normal variant.

In general, to use the reciprocal variant of a CRC-model, following changes are required for the CRC model: The CRC parameters *Polynomial* and *Initial Value* have to be reflected, the parameters *Input reflected* and *Result reflected* have to be negated and the parameter *Final XOR value* remains unchanged. Here some code to change the CRC model to the reciprocal CRC model:

```
public static Crc32Model GetReflectedCrcModel(Crc32Model model)
{
    return new Crc32Model(
        CrcUtil.Reflect32(model.Polynomial),
        CrcUtil.Reflect32(model.Initial),
        model.FinalXor,
        !model.InputReflected,
        !model.ResultReflected
    );
}
```

With this information, here the table-based calculation of reciprocal CRC-32 (right) compared to the normal calculation (left):

```
public uint Compute(byte[] bytes)
{

    uint crc = crcModel.Initial;
    foreach (byte b in bytes)
    {
        byte curByte = (crcModel.InputReflected
? CrcUtil.Reflect8(b) : b);
```

```
public uint ComputeReciprocal(byte[] bytes)
{
    crcModel = GetReflectedCrcModel(crcModel);
    uint crc = crcModel.Initial;
    foreach (byte b in bytes)
    {
        byte curByte = (crcModel.InputReflected
? CrcUtil.Reflect8(b) : b);
```

```
        /* update the MSB of crc value with
next input byte */
        crc = (uint)(crc ^ (curByte << 24));
        /* this MSB byte value is the index
into the lookup table */
        byte pos = (byte)(crc >> 24);
        /* shift out this index */
        crc = (uint)(crc << 8);
        /* XOR-in remainder from lookup table
using the calculated index */
        crc = (uint)(crc ^
(uint)crcTable[pos]);
    }
    crc = (crcModel.ResultReflected ?
CrcUtil.Reflect32(crc) : crc);
    return (uint)(crc ^ crcModel.FinalXor);
}
```

```
        /* update the LSB of crc value with
next input byte */
        crc = (uint)(crc ^ curByte);
        /* this byte value is the index into
the lookup table, make sure it's a byte */
        byte pos = (byte)(crc & 0xFF);
        /* shift out this index */
        crc = (uint)(crc >> 8);
        /* XOR-in remainder from lookup table
using the calculated index */
        crc = (uint)(crc ^
(uint)crcTable[pos]);
    }
    crc = (crcModel.ResultReflected ?
CrcUtil.Reflect32(crc) : crc);
    return (uint)(crc ^ crcModel.FinalXor);
}
```

7.4 Extending to arbitrary CRC sizes (width of polynomials)

[Back to top]

Until now, only CRC widths of 8, 16 and 32 have been considered, thus integer multiples of byte (8 bit) sizes. Of course, there are also CRC algorithms with other widths. What needs to be considered in the implementation to support those kinds of CRC algorithms? This is discussed in this chapter. Note that in the following, still the input values are assumed to be bytes (8 bit values).

It is obvious that the underlying data type must be sufficiently large to hold the values (so e.g. use 32 bit values for CRC-28). Considering the simple bitwise implementation, following values depend on the CRC width and needs to be adapted as follows (Note: At first, only CRC with > 8 are considered):

CRC > 8:

- The number of bits to move the next byte into the MSB of the CRC register depends on the CRC width. E.g. for CRC-11, the input byte needs only be shifted by 3.
  In general: Number of bits to left shift = crcsize - 8.
- The most significant bit of the CRC is checked if it is set. Therefore the needed bit mask depends on the CRC width. E.g. a CRC-11 value has 11 binary digits, so the bit mask is in binary b100 0000 0000 = 0x400.
  In general: topbitmask = 1 << (crcsize - 1).
- For a CRC-x, only the lower x bits are significant. If a larger data type is used, then the upper unused bits shall be cleared, otherwise the result is interpreted incorrectly. So a mask should be applied to the CRC value. E.g for CRC-11, only the 11 lower bits are relevant, thus the mask would be b111 1111 1111 = 0x7FF.
  In general: final mask = (1 << crcsize) - 1.
- The reflection of a value must also be considered.

With this information, a sample CRC implementation for CRC width of 8 or greater might look like the following (Note: due to usage of ushort, actually only up to CRC-16 is supported but this changed by replacing uhsort by a larger data type):

```
public static ushort Compute_Simple_Crc_8_16(byte[] bytes)
{
    ushort poly = 0x307;           // setup example poly
    ushort crcsize = 11;           // setup example crc width
    ushort initialValue = 0;       // setup example initial value
    bool inputReflected = false;   // setup example input reflection
    bool resultReflected = false;  // setup example result reflection
    ushort finalXor = 0;           // setup example final XOR value

    ushort bitsToShift = (ushort)(crcsize - 8);       // example CRC-11: 3
    ushort bitmask = (ushort)(1 << (crcsize - 1));    // example CRC-11: 0x0400
    ushort finalmask = (ushort)((1 << crcsize) - 1);  // example CRC-11: 0x07FF

    ushort crc = initialValue;

    foreach (byte b in bytes)
    {
```

```
        byte curByte = (inputReflected ? Reflect8(b) : b); ;
        crc ^= (ushort)(curByte << bitsToShift); /* move byte into MSB of CRC */

        for (int i = 0; i < 8; i++)
        {
            if ((crc & bitmask) != 0)
            {
                crc = (ushort)(((crc << 1) ^ poly));
            }
            else
            {
                crc <<= 1;
            }
        }

        crc = (ushort)(crc & finalmask);
    }

    crc = (ushort)(resultReflected ? ReflectGeneral(crc, crcsize) : crc);
    crc ^= finalXor;

    return (ushort)(crc);
}
```

CRC < 8:

For a CRC width smaller than 8, additional points must taken into account because byte values are used as input values. This means the CRC register is actually smaller than an input value. This is no issue when using the initial shift register implementation. However, it matters for our simple implementation where whole bytes are "xored" into the CRC register. It would be required to only "xor" the relevant bits and handle the remaining bits of the input byte in the next loop which complicates the handling of the inputs bytes significantly.

An alternative approach which results in a simpler implementation is to use always an 8 bit CRC register also for a CRC width smaller than 8. The trick is then to only work the "x" most significant bits of the CRC register for a CRC with width "x". This means:

- The next byte of the input byte stream is just moved into the CRC register, no need to shift the value inside the CRC register.
- The top bit mask is always 0x80.
- The final mask is always 0xFF.
- The initial and polynomial values are shifted to the left so that both are located in the most signficant bits of the 8-bit CRC value.
- At the end of the calculation, the CRC value is shifted back so that it is located in the least signficant bits of the 8-bit CRC register value, otherwise the result is interpreted incorrectly.

Summarized, a possible implementation for CRC-x (where x < 8) might look like following:

```
public static ushort Compute_Simple_Crc_SmallerThanEight(byte[] bytes)
{
    byte poly = 0x09;             // setup example poly
    byte crcsize = 7;              // setup example crc width
    byte initialValue = 0;         // setup example initial value
    bool inputReflected = false;   // setup example input reflection
    bool resultReflected = false;  // setup example result reflection
    byte finalXor = 0;              // setup example final XOR value

    byte bitmask = 0x80;
    byte finalmask = 0xFF;

    byte bitOffsetTo8 = (byte)(8 - crcsize);
    poly <<= bitOffsetTo8;

    ushort crc = initialValue;
    crc <<= bitOffsetTo8;

    foreach (byte b in bytes)
    {
        byte curByte = (inputReflected ? Reflect8(b) : b); ;
        crc ^= (ushort)(curByte); /* move byte into MSB of CRC */

        for (int i = 0; i < 8; i++)
        {
```

```
            if ((crc & bitmask) != 0)
            {
                crc = (ushort)(((crc << 1) ^ poly));
            }
            else
            {
                crc <<= 1;
            }
        }

        crc = (ushort)(crc & finalmask);
    }

    crc >>= bitOffsetTo8;

    crc = (ushort)(resultReflected ? ReflectGeneral(crc, crcsize) : crc);
    crc ^= finalXor;

    return (ushort)(crc);
}
```

8. Additional remarks (points worth to know)

[Back to top]

This last chapter contains interesting (and maybe not completely obvious) topics about CRC calculation. These points are optional and contain just additional information, for those who are interested in same background information.

8.1 Basic mathematical view of CRC (read it first)

[Back to top]

First let's recall some mathematical basics for CRC definition for better understanding of the next points. Please note that the sub chapter actually represents the same information as chapter 2 (don't hesitate to have a quick look back at this introduction chapter):

CRC-n uses a generator polynomial G(x) which has degree n and n+1 terms: $x^n + x^{n-1} + ... + x^1 + x^0$.
The n+1 terms means that the polynomial has a length of n + 1 bits (using normal representation the most significant bit is left out).

> Example: A CRC-8 with polynomial 0x07 is actually the value 100000111 = $1*x^8 + 0*x^7 + 0*x^6 + 0*x^5 + 0*x^4 + 0*x^3 + 1*x^2 + 1*x^1 + 1*x^0$.

The computation of CRC, which we know is based on polynomial division (or more specific on *Polynomial arithmetic modulo 2*), can be stated as $M(x) * x^n = G(x) * Q(x) + R(x)$ where:

- **M(x)** is the input binary string, so $M(x)*x^n$ is the input string with n zero bits appended. (Stop! Why is it like that? Answer: We'll come to this later in chapter 8.5).
- **G(x)** is the generator polynomial with degree n as already stated.
- **Q(x)** is quotient of the division and not used further.
- **R(x)** is the remainder = the actual CRC value.

8.2 Background to CRC verification

[Back to top]

Or: "Why is the verification result zero if the CRC is computed over the input data with the actual CRC value appended (in the manual approach)?"

Actually, the XOR arithmetic of CRC division is comparable to school arithmetic in this case, so let's start with an example.

> Assume the input data $M(x) * x^n$ is the number 195 and the divisor G(x) is 29. We would calculate:
>
> 195 / 29 = 6 remainder 21(side note 1). This is the same as 195 = 29 * 6 + 21 (which corresponds to $M(x) * x^n = G(x) * Q(x) + R(x)$). So the CRC value R(x) is 21.

First verification approch (CRC is transmitted separately to the input data):
The sender sends the input data 195 along with the CRC value 21. The receiver then would take the input data and perform the same calculation (remember the generator polynomial is fixed and statically known by transmitter and receiver): 195 % 29 = 21. The calculated remainder is the same as the received one, so the data transmission was faultless.

Second verification approach (CRC is appended to the input data):
The CRC value is appended to the input data which corresponds in school arithemtic to subtraction. So the CRC value 21 is subtracted from input data 195 resulting in 174 (= M(x) * x^n - R(x)). The receiver would perform 174 % 29 = 0, so the remainder is zero and the data transmission was faultless.

We know that M(x) * x^n is the input data bit string with n zero bits appended. As we see later (in chapters 8.4 and 8.5) appending the bits of R(x) to this input string can be stated as M(x) * x^n - R(x).

Rearrange the formula: M(x) * x^n = G(x) * Q(x) + R(x) ===> M(x) * x^n - R(x) = G(x) * Q(x)

Here we see that M(x) * x^n - R(x) (which is the input data with the CRC appended) is an integer multiple of the polynomial G(x). And this means that M(x) * x^n - R(x) divided by G(x) results in 0.

Side note 1: M(x) * x^n = G(x) * Q(x) + R(x) can also be written using the modulo operator *mod* as R(x) = (M(x) * x^n) mod G(x).

8.3 CRC-1 is the same as a parity bit

[Back to top]

This is correct. Here some clarifying words: A parity bit indicates if the number of bits with value 1 is even or odd. In the case of even parity, the number of bits whose value is 1 in a given set are counted. If that total is odd, the parity bit value is set to 1, making the total count of 1's in the set an even number.

Example: The value 0x34 = 0011 0100 has three '1' bits. Because three is odd, the parity bit is 1.

CRC-1 has degree 1 and 2 terms: a*x^1 + b*x^0. The most significant bit is always 1. However a polynomial 10 has no sense because the actual CRC value would always be 0. So CRC-1 has polyomial 11 (binary) which is just 0x01 in normal representation. And this is in fact the calculation of an even parity bit. Remember that the actual CRC value has n bits, so for CRC-1 the remainder has 1 bit, either 0 or 1.

Here an example for the value 0x34:

```
001101000
  11
---------
  0001000
     11
  -------
     0100
      11
      ---
      010
       11
       --
       01 = CRC = 1
```

8.4 Why is addition is the same as subtraction in CRC arithmetic?

[Back to top]

CRC computation is performed using so-called polynomial arithmetic. This polynomial arithmetic is based on division over the finite field with two elements: 0 and 1. Also called Galois field over two elements.
To define the addition operation, there are only four cases to distinct:
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 - note that there is no carry!

Due to the fact that we perform the calculation over a finite field (refer to [6] for more info), there is only one way to define subtraction:
0 - 0 = 0
0 - 1 = 1
1 - 0 = 1
1 - 1 = 0

We see that addition and subtraction are the same: Basically this is the XOR operation we have already used several times above. That's the reason why we stated above that M(x) * x^n - R(x) and M(x) * x^n + R(x) are the same in CRC arithmetic.

8.5 Why does multiplication with x^n append n zeros?

[Back to top]

This is due to the fact that each trailing zero bit of adds a factor of x. Let's say we have the polynomial $x^n + x^{n-1} + ... + x^1 + x^0$. Multiply it with x:
$x* (x^n + x^{n-1} + ... + x^1 + x^0) = x^{n+1} + x^n + ... + x^1$
So by multiplying with x, we have increased the degree of the polynomial by 1, which equals appending a zero to the right. Maybe this gets clearer with an example:

Example:

Assume the polynomial 01010010 = x^6 + x^4 + x and add a zero:
010100100 = x^7 + x^5 + x^2 = **x** * (x^6 + x^4 + x). So each factor of x appends a zero.

8.6 When using an initial value other than zero in the shift register, the result is incorrect.

[Back to top]

In chapter 7 initial values were introduced, with the description that it's the value used to initialize the shift register. Well, you can read this statement also in other CRC articles, however there is a trap that is commonly not well described (if at all).

Example: If that would be true than e.g. following two CRC instances should calculate the same CRC:

Instance a: Initial value = 0x00, polynomial = 0x9B, input message data = [0xFF, 0x01].
Instance b: Initial value = 0xFF, polynomial = 0x9B, input message data = [0x01].

Assumption: Because in CRC instance a), the initial value is zero, nothing will happen (meaning no xor operation will be computed), until the first data byte 0xFF is completely shifted into the register, because 0x00 has no bit set. Then we have actually the same state as CRC instance b), so the processing of the remaining byte 0x01 should result the in the same CRC value for both cases. Right?

Unfortunately, not. Above examples give the two different CRC values of 0x2A and 0xE0.

Looking at the simple (non-lookup table, but byte-wise handling) CRC-8 implementation, we see the reason:

```
public byte Compute_Simple(byte[] bytes)
{
  byte crc = crcModel.Initial;

  foreach (byte b in bytes)
  {
    byte curByte = (crcModel.InputReflected ? CrcUtil.Reflect8(b) : b);
    crc ^= curByte; /* XOR-in the next input byte */

    for (int i = 0; i < 8; i++)
    {
      if ((crc & 0x80) != 0)
      {
          crc = (byte)((crc << 1) ^ crcModel.Polynomial);
      }
      else
      {
          crc <<= 1;
      }
    }
  }
  crc = (crcModel.ResultReflected ? CrcUtil.Reflect8(crc) : crc);
  return (byte)(crc ^ crcModel.FinalXor);
}
```

> Before the actual first xor operation of the CRC calculation is performed, the initial value is xored with the first input byte!

What does that mean for the initial value of the shift register when performing the CRC calculation bit by bit, like the pen and paper approach at the top? Answer: The register must be initialized with the initial value 'xored' with the first input bytes. For CRC-16 the initial value must be xored with the first two input bytes, for CRC-32 with the first four input bytes, and so on.
Here an example implementation how to initialize a CRC-32 shift register and the corresponding shift register algorithm. This is taken from the C# source package linked at the top of the article:

```
private uint GetInitialShiftRegister(byte[] bytes)
{
  byte b1 = 0, b2 = 0, b3 = 0, b4 = 0;
  if (bytes.Length >= 1)
  {
    b1 = crcModel.InputReflected ? CrcUtil.Reflect8(bytes[0]) : bytes[0];
  }
  if (bytes.Length >= 2)
  {
    b2 = crcModel.InputReflected ? CrcUtil.Reflect8(bytes[1]) : bytes[1];
  }
  if (bytes.Length >= 3)
  {
    b3 = crcModel.InputReflected ? CrcUtil.Reflect8(bytes[2]) : bytes[2];
  }
  if (bytes.Length >= 4)
  {
    b4 = crcModel.InputReflected ? CrcUtil.Reflect8(bytes[3]) : bytes[3];
  }
  return (uint)(crcModel.Initial ^ (uint)((uint)b4 << 24 | (uint)b3 << 16 | (uint)b2 << 8 |
b1));
}

public uint Compute_Simple_ShiftReg(byte[] bytes)
{
    uint crc = GetInitialShiftRegister(bytes);
    /* skip first four bytes, already inside crc register */
    for (int byteIndex = 4; byteIndex < bytes.Length + 4; byteIndex++)
    {
      byte curByte = (byteIndex < bytes.Length) ? (crcModel.InputReflected ?
CrcUtil.Reflect8(bytes[byteIndex]) : bytes[byteIndex]) : (byte)0;

      for (int i = 0; i <= 7; i++)
      {
        if ((crc & LOWBIT_MASK) != 0)
        {
          crc = (uint)(crc >> 1);
```

```
          crc = ((uint)(curByte & (1 << i)) != 0) ? (uint)(crc | 0x80000000) : (uint)(crc &
0x7FFFFFFF);
          crc = (uint)(crc ^ crcModel.Polynomial);
        }
        else
        {
          crc = (uint)(crc >> 1);
          crc = ((uint)(curByte & (1 << i)) != 0) ? (uint)(crc | 0x80000000) : (uint)(crc &
0x7FFFFFFF);
        }
      }
    }
    crc = (crcModel.ResultReflected ? CrcUtil.Reflect32(crc) : crc);
    return (uint)(crc ^ crcModel.FinalXor);
}
```

9. Conclusion & References

[Back to top]

This article is my own guide to CRC: very practical and based on many examples - quite different to most of the other articles on the web. I hope you found it interesting and that it supported you in understanding and implementing CRC!
Drop me a line for corrections, hints, criticism, praise ;)

Sunshine, February 2015 (last update: June 2023)

References

[1] Cyclic redundancy check @ Wikipedia
[2] "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks". The International Conference on Dependable Systems and Networks: 145–154. Retrieved 29 December 2014.
[3] Mathematics of cyclic redundancy checks @ Wikipedia
[4] A Painless Guide to CRC Error Detection Algorithms
[5] CRC RevEng - Catalogue of parametrised CRC algorithms
[6] Finite field @ Wikipedia

Updates

- 2023/06/19:
  - Fixed grammar: Replaced divident by dividend.
  - Clarified several locations that CRC-n uses a polynomial of degree n which has n+1 terms.
- 2023/03/11:
  - Added chapter 7.4 (Extending to arbitrary CRC sizes (width of polynomials))
- 2019/03/22:
  - Fixed a bug in chapter 7.3.1 about the final XOR value when using the reciprocal variant: In the reciprocal CRC model, the final XOR value *must not* be reflected, contrary to my previous statement.
    Note that in the source code, this was already correctly implemented.
- 2018/12/29:
  - Replaced occurrences of x^n by x^n for consistent naming.
- 2018/09/29:
  - Fixed a glitch in the "Step-by-step visualization of simple CRC-8 algorithmus" table. Thanks to the careful reader for pointing me to it.
  - Reworked the chapters of representation of polynomials and reciprocal polynomials - hopefully they are more understandable now (chapter 7.2 and 7.3).
  - Fixed a typo in chapter 8.6 (byte 0x01 instead of 0x11). Thanks to the careful reader for pointing me to it.
- 2016/11/11:
  - Added chapters 7.3 (reversed polynomials) and chapter 8.6 (initial value for shift register)

    - Updated C# source package with CRC classes that work in the reversed way.
    - Updated Javascript Online calculator to show optionally the CRC lookup table in the reversed way.
- 2016/08/19: Fixed a typo in chapter 8.4: 1 + 1 = 0 instead of 1 + 1 = 1.
- 2015/05/30: Added chapter 8.

This site is part of Sunshine's Homepage