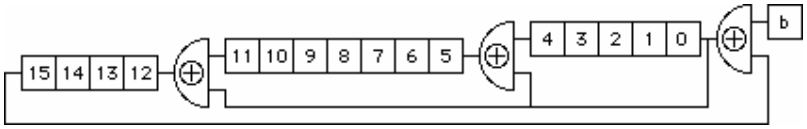
Cyclic Redundancy Check

Introduction

Different methods exist to calculate a check number for binary data, to be able to see if the data is not altered, for example, after being sent through some communication channel. Cyclic Redundancy Check (CRC) is a common method for protecting binary data that way. Different CRCs exist, which in the past has resulted in a naming scheme. This page discusses the standard ITU-TSS CRC, often written as a formula: $G(x)=x^{16}+x^{12}+x^{5}+1$.
Characteristic of this CRC is its 16 bits size and its initial value $FFFF. Although, you can encounter an initial value $0000 too.

Hardware

A CRC generator can be built as a piece of hardware.

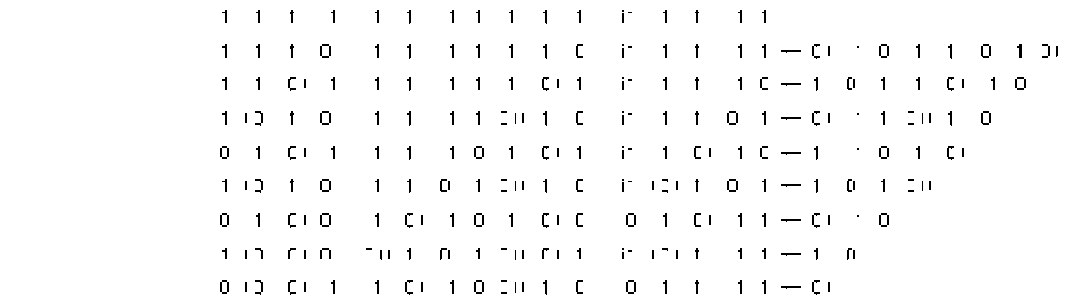


Each bit (b) of the binary data is shifted into the CRC register after being XORed with the CRCs most significant bit. This part of the generator ensures the cyclical aspect of CRC. The XOR result is inserted in CRC bit 5 and 12 too. During the processing of bit b, all current CRC bits (modified or unmodified) are shifted one position to the left.
If a byte must be processed, all 8 bits must be processed one after another. The most significant bit is processed first.

Example:

Initial CRC value: 1111 1111 1111 1111

Byte to process: 0101 1010



New CRC value: 0001 1010 0100 1111

Formula

Expressed as a set of formulas, processing a bit b means:

$C^*_{15}=C_{14}$ $C^*_{14}=C_{13}$ $C^*_{13}=C_{12}$ $C^*_{12}=C_{11}\oplus C^\wedge_0$

$C^*_{11}=C_{10}$ $C^*_{10}=C_9$ $C^*_9=C_8$ $C^*_8=C_7$

$C^*_7=C_6$ $C^*_6=C_5$ $C^*_5=C_4\oplus C^\wedge_0$ $C^*_4=C_3$

$C^*_3=C_2$ $C^*_2=C_1$ $C^*_1=C_0$ $C^*_0=C_{15}\oplus b$

Software

One can build a CRC generator in software, that is analoguous to the hardware solution. The solution processes each bit separately.
Using the C programming language, the source code could be:

```
unsigned short crc = 0xFFFF;
unsigned short temp;
unsigned char byte = 0x5A; //just as an example
unsigned short index;

for(index = 0; index <= 7; index++)
   {
   temp = (crc >> 15) ^ (byte >> 7);
   crc <<= 1;
   if(temp)
      {
      crc ^= 0x1021;
      }
   byte <<= 1;
   }
```

First, b XOR c_{15} (the cyclical value) is calculated. Then the CRC bits are shifted to the left (c_0 becomes 0). The cyclical value has to be processed in c_0, c_5 and c_{12}. If the cyclical value equals 1, bits c_0, c_5 and c_{12} are changed at the same moment by XORing the CRC with value 0001 0000 0010 0001 (0x1021). If the cyclical value equals 0, the CRC should be XORed with value 0000 0000 0000 0000. XORing the CRC with 0x0000 does not change the CRC, and therefore it is skipped. Finally the next bit to be processed is prepared.
In the source code example, the initial CRC value and the byte to be processed are already defined. The calculations sub results are:

crc	byte
F F F F	5 A
E F D F	B 4
D F B E	6 8
A F 5 D	D 0
5 E B A	A 0
A D 5 5	4 0
4 A 8 B	8 0
8 5 3 7	0 0
1 A 4 F	

Simplification

When processing a byte, the eight most significant bits of the 'old' CRC value, will be shifted out of the CRC. Somewhere in the process, these bits will be XORed with the byte to be processed. This XOR can be done at once and as a first step. A temporary variable is not needed too.
The simpified source code could be:

```
unsigned short crc = 0xFFFF;
unsigned char byte = 0x5A;
unsigned short index;

crc ^= byte << 8;
for(index = 0; index <= 7; index++)
  {
  crc = crc & 0x8000 ? (crc << 1) ^ 0x1021 : crc << 1;
  }
```

In the source code example too, the initial CRC value and the byte to be processed are already defined. The calculations sub results are:

crc

A 5 F F

5 B D F

B 7 B E

7 F 5 D

F E B A

E D 5 5

C A 8 B

8 5 3 7

1 A 4 F

Acceleration

To be able to accelerate the software CRC generator, we first take a look at the formulas that express the processing of 8 bits ($b_7..b_0$), combined into a byte ($b_{7..0}$).

$$t_{7..0} = c_{15..8} \oplus b_{7..0}$$

$$t_{7..4}$$

$$c_{15..13} = c_{7..5} \oplus t_{3..1} \oplus t_{7..5}$$

$$c_{12..12} = c_{4..4} \oplus t_{0..0} \oplus t_{4..4} \oplus t_{7..7}$$

$$c_{11..9} = c_{3..1} \oplus t_{6..4}$$

$c_{8..8} = c_{0..0} \oplus t_{3..3} \oplus t_{7..7}$

$c_{7..5} = t_{7..5} \oplus t_{2..0} \oplus t_{6..4}$

$c_{4..4} = t_{4..4}$

$c_{3..0} = t_{3..0} \oplus t_{7..4}$

Note that combination $t_{3..0} \oplus t_{7..4}$ is used serveral times (when you combine the proper formula lines). The same way, $\mathbf{t_{7..0}}$ can be found a few times too. We use this fact to simplify the formulas.

$t_{7..0} = c_{15..8} \oplus b_{7..0}$

$c_{15..0} = c_{7..0} \mid 00000000$

$q_{15..0} = 00000000 \mid t_{7..0} \oplus 000000000000 \mid t_{7..4}$

$c_{15..0} = c_{15..0} \oplus q_{15..0}$

$q_{15..0} = 000 \mid q_{7..0} \mid 00000$

$c_{15..0} = c_{15..0} \oplus q_{15..0}$

$q_{15..0} = q_{8..5} \mid 000000000000$

$c_{15..0} = c_{15..0} \oplus q_{15..0}$

A | means the bit parts have to combined into one value.
The simplified formulas process 8 bits at a time. They form the basis of an accelerated source code example.

```
unsigned short crc = 0xFFFF;
unsigned short temp;
unsigned short quick;
unsigned char byte = 0x5A;

temp = (crc >> 8) ^ byte;
crc <<= 8;
quick = temp ^ (temp >> 4);
crc ^= quick;
quick <<= 5;
crc ^= quick;
quick <<= 7;
crc ^= quick;
```

In this source code example too, the initial CRC value and the byte to be processed are already defined. The one step calculation result is:

crc	byte
F F F F	5 A
1 A 4 F	

32-bit CRC

The ITU-TSS has defined a 32-bit CRC too. Its formula is:
$G(x)=x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1=0$
Below is a source code example for calculating the 32-bit CRC.

```
unsigned long crc = 0xFFFFFFFF;
unsigned char byte = 0x5A;
unsigned short index;

crc ^= byte << 24;
for(index = 0; index <= 7; index++)
   {
   crc = crc & 0x80000000 ? (crc << 1) ^ 0x04C11DB7 : crc << 1;
   }
```

2003-11-19