



	<h1>CRC and how to Reverse it</h1> <h2>A CRC Tutorial & The c00l way to Reverse CRC</h2>	
Release: 29 april 1999 Modified: 30 april 1999	by anarchriz	
	Courtesy of Fravia's page of reverse engineering	slightly edited by Fravia+
fra_00xx 990504 anarchriz 1100 PA PC	<i>A beautiful study, that rightly belongs to the +HCU Papers</i>	
	There is a crack, a crack in everything That's how the light gets in	
Rating	<input checked="" type="checkbox"/> Beginner <input checked="" type="checkbox"/> Intermediate <input checked="" type="checkbox"/> Advanced <input type="checkbox"/> Expert	

You always wanted to know what CRC exactly is? Always wanted to know how to compute it yourself? Ever thought about ways to reverse CRC, but didnt succede? Ever tried to patch a piece of code without altering its CRC? Ever wanted to write an anti-antivirus trick to render the CRC32 check useless? Well then... you may have landed at the right place!

CRC and how to Reverse it

A CRC Tutorial & The c00l way to Reverse CRC

Written by [anarchriz](#)

Introduction

This essay consists of a CRC tutorial and a way of how to reverse it. Many Coders/Fravias don't know exactly how CRC works and almost no one knows how to reverse it, while this knowledge could be very usefull. First the tutorial will learn you how to calculate CRC in general, you can use it as data/code protection. Second, the reverse part will learn you (mainly) how to reverse CRC-32, you can use this to break certain CRC protections in programs or over programs (like anti-virus). There seem to be utilities who can 'correct' CRCs for you, but I doubt they also explain what they're doing.

I'd like to warn you, since there is quite some math used in this essay. This wont harm anyone, and will be well understood by the avarage Fravia or Coder. Why? Well. If you dont know why math is used in CRC, I suggest that you click that button with a X at the top-right of this screen. So I assume the reader has knowledge of binair arithmetic.

Essay

Part 1: CRC Tutorial, what it is and how to calculate it

Cyclic Redundancy Code or CRC

We all know CRC. Even if you don't recall, you will when you think of those annoying messages RAR, ZIP and other compressors give you when the file is corrupted due to bad connections or those !@#\$\$ floppies. The CRC is a value computed over a piece of data, for example for each file at the time of compression. When the archiver is unpacking that file, it will read the CRC and check it with the newly computed CRC of the uncompressed file. When they match, there is a good chance that the files are identical. With CRC-32, there is a chance of $1/2^{32}$ of the check failing to recognize a change in data.

A lot of people think CRC is short for Cyclic Redundancy Check. If indeed CRC is short for Cyclic Redundancy Check then a lot of people use the term incorrect. If it was you could not say 'the CRC of the program is 12345678'. People are also always saying a certain program has a CRC check, not a Cyclic Redundancy Check check. Conclusion: CRC stands for Cyclic Redundancy Code and NOT for Cyclic Redundancy Check.

How is the calculation done? Well, the main idea is to see the file as one large string of bits divided by some number, which will leave you with a remainder, the CRC! You always have a remainder (can also be zero) which is at most one bit less than the divisor (else it still has a divisor in it).

($9/3=3$ remainder=0 ; ($9+2$)/3=3 remainder=2)

Only here dividing with bits is done a little different. Dividing is repeatedly subtracting (x times) a number (divisor) from a number you want to divide, which will leave you with the remainder. If you want the original number back you multiply with the divisor or (idem) add x times the divisor with itself and afterwards adding the remainder.

CRC computation uses a special way of subtracting and adding, i.e. a new 'arithmetic'. While computing the carry for each bit calculation is 'forgotten'.

Lets look at 2 examples, number 1 is a normal subtraction, 2&3 are special.

```
-+
(1) 1101 (2) 1010 1010 (3) 0+0=0 0-0=0
    1010-      1111+ 1111-      0+1=1 *0-1=1
    ----      ----  ----      1+0=1 1-0=1
    0011      0101 0101      *1+1=0 1-1=0
```

In (1), the second column from the right would evaluate to $0-1=-1$, therefore a bit is 'borrowed' from the bit next to it, which will give you this subtraction $(10+0)-1=1$. (this is like normal 'by-paper' decimal subtraction) The special case (2&3) $1+1$ would normally have as answer 10, where the '1' is the carry which 'transports' the value to the next bit computation. This value is forgotten. The special case $0-1$ would normally have as answer '-1', which would have impact on the bit next to it (see example 1). This value is also forgotten. If you know something about programming this looks like, or better, it IS the XOR operation.

Now look at an example of a divide:

In normal arithmetic:

```
1001/1111000\1101 13      9/120\13
 1001  -                09  -|
 ----                --  |
 1100                  30  |
 1001  -                27  -
 ----                --
 0110                  3 -> the remainder
 0000  -
 ----
 1100
 1001  -
 ----
 011 -> 3, the remainder
```

In CRC arithmetic:

1001/1111000\1110 9/120\14 remainder 6

```

1001   -
----
1100
1001   -
----
1010
1001   -
----
0110
0000   -
----
110 -> the remainder
(example 3)
```

The quotient of a division is not important, and not efficient to remember, because that would be only a couple of bits less than the bitstring where you wanted to calculate the CRC from. What IS important is the remainder! That's the thing that says something important over about the original file. That's basically the CRC!

Going over to the real CRC computation

To perform a CRC calculation we need to choose a divisor, we call it the 'poly' from now on. The width W of a poly is the position of the highest bit, so the width of poly 1001 is 3, and not 4. Note that the highest bit is always one, when you have chosen the width of the poly you only have to choose a value for the lower W bits.

If we want to calculate the CRC over a bitstring, we want to make sure all the bits are processed. Therefore we need to add W zero bits to the end of the bitstring. In the case of example 3, we could say the bitstring was 1111. Look at a little bigger example:

Poly = 10011, width $W=4$
Bitstring + W zeros = 110101101 + 0000

10011/1101011010000\110000101 (we don't care about the quotient)

```

10011| | | | | -
-----| | | | |
10011| | | | |
10011| | | | | -
-----| | | | |
00001| | | | |
00000| | | | | -
-----| | | | |
00010| | | | |
00000| | | | | -
-----| | | | |
00101| | | | |
00000| | | | | -
-----| | | | |
01010| | | | |
00000| | | | | -
-----| | | | |
10100| | | | |
10011| | | | | -
-----| | | | |
01110| | | | |
00000| | | | | -
-----| | | | |
11100
10011   -
-----
1111 -> the remainder -> the CRC!
(example 4)
```

There are 2 important things to state here:

1. Only when the highest bit is one in the bitstring we XOR it with the poly, otherwise we only 'shift' the bitstring one bit to the left.
2. The effect of XORring is, that it's XORred with the lower W bits, because the highest bit always gives zero.

Going over to a Table-Driven Algorithm

You all should understand that an algorithm based on bitwise calculation will be very slow and inefficient. It would be far more efficient if you could calculate it on a per-byte basis. But then we can only accept poly's with a width of a multiple of 8 bits (that's a byte ;). Lets visualize it in a example poly with a width of 32 (W=32):

```

      3   2   1   0   byte
    +---+---+---+---+
Pop! <--|   |   |   |   | <-- bitstring with W zero bits added, in this case 32
    +---+---+---+---+
    1<--- 32 bits ---> this is the poly, 4*8 bits

```

(figure 1)

This is a register you use to store the temporary result of the CRC, I call it the CRC register or just register from now on. You are shifting bits from the bitstring in at the right side, and bits out at the left side. When the bit just shifted out at the left side is one, the whole register is XORred by the lower W bits of the poly (in this case 32). In fact, we are doing exactly the same thing as the divisions above. What if (as I said) we would shift in & out a whole group of bits at once. Look at an example of 8 bit CRC with 4 bits at once shifted in & out:

The register just before the shift : 10110100
Then 4 bits (at the top) are shifted out at the left side while shifting 4 new bits in at the right side. In this example 1011 is shifted out and 1101 (new) is shifted in.

Then the situation is this:
8 bits currently CRC/Register : 01001101
4 top bits just shifted out : 1011
We use this poly : 101011100, width W=8

Now we calculate just as usual the new value of the register.

```

Top Register
-----
1011 01001101 the topbits and the register
1010 11100 + (*1) Poly is XORred on position 3 of top bits (coz there is a one)
-----
0001 10101101 result of XORring

```

Now we still have a one on bit position 0 of topbits:
0001 10101101 previous result
1 01011100+ (*2) Poly is XORred on position 0 of top bits (coz there is a one)

0000 11110001 result of second XORring
^^^^

Now there are all zero's in the topbits, so we dont have to XOR with the poly anymore for this sequence of topbits.

The same value in the register you get if you first XOR (*1) with (*2) and the result with the register. This is because of the standard XOR property:
(a XOR b) XOR c = a XOR (b XOR c)

```

1010 11100 poly on position 3 of top bits
1 01011100+ poly XORred on position 0 of top bits
-----

```

1011 10111100 (*3) result of XORring

The result (*3) is XORred with the register
1011 10111100
1011 01001101+ the top bits and the register

0000 11110001

You see? The same result! Now (*3) is important, because with the top bits 1010 is always the value (*3)=10111100 (only the lower W=8 bits) bound (under the stated conditions, of course) This means you can precompute the XOR values for each combination of top bits. Note that top bits always become zero after one iteration, this must be because the combination of XORring leads to it.

Now we come back to figure 1. For each value of the top byte (8 bits) just shifted out, we can precompute a value. In this case it would be a table consisting of 256 (2^8) entries of double words (32bit). (the CRC-32 table is in the appendix)

In pseudo-language our algorithm now is this:

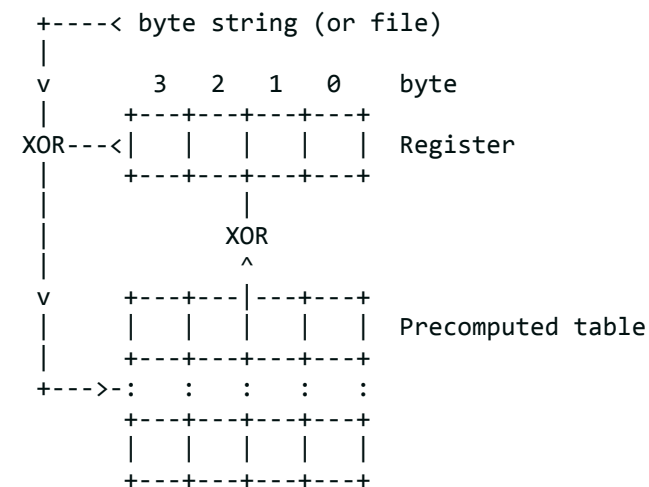
```
While (byte string is not exhausted)
  Begin
    Top = top_byte of register ;
    Register = Register shifted 8 bits left ORred with a new byte from string ;
    Register = Register XORred by value from precomputedTable at position Top ;
  End
```

The direct Table Algorithm

The algorithm proposed above can be optimized. The bytes from the byte string don't need to travel through the whole register before they are used. With this new algorithm we can directly XOR a byte from a byte string with the byte shifted out of the register. The result points to a value in the precomputed table which will be XORred with the register.

I don't know exactly why this gives the same result (it has to do with a XOR property), but it has the Big advantage you don't have to append zero bytes/bits to your byte string. (if you know why, pleaz tell me :)

Lets visuallize this algorithm:



(figure 2)

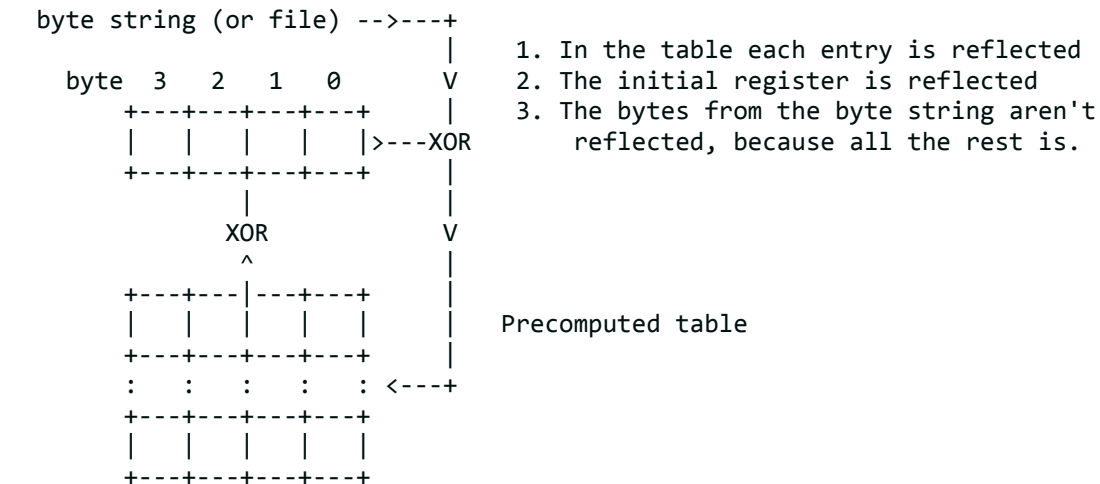
The 'reflected' direct Table Algorithm

To make things more complicated there is a 'reflected' version of this algorithm. A Reflected value/register is that it's bits are swapped around it's centre. For example 0111011001 is the reflection of 1001101110.

They came up with this because of the UART (chip that performs serial IO), which sends each byte with the least significant bit (bit 0) first and the most significant bit (bit 7) last, this is the reverse of the normal situation.

Instead then of reflecting each byte before processing, every else is

reflected. An advantage is that it gives more compact code in the implementation. So, in calculating the table, bits are shifted to the right and the poly is reflected. In calculating the CRC the register is shifted to the right and (of course) the reflected table is used.



(figure 3)

Our algorithm is now:

1. Shift the register right by one byte
2. XOR the top byte just shifted out with a new byte from the byte string to yield an index into the table ([0,255])
3. XOR the table value into the register
4. Goto 1 if there are more bytes to process

Some implementations in Assembly

To get everything settled here's the complete CRC-32 standard:

```
Name      : "CRC-32"
Width     : 32
Poly      : 04C11DB7
Initial value : FFFFFFFF
Reflected : True
XOR out with : FFFFFFFF
```

As a bonus for you curious people, here's the CRC-16 standard: :)

```
Name      : "CRC-16"
Width     : 16
Poly      : 8005
Initial value : 0000
Reflected : True
XOR out with : 0000
```

'XOR out with' is the value that is XORred with the final value of the register before getting (as answer) the final CRC.

There are also 'reversed' CRC poly's but they are not relevant for this tutorial. Look at my references if you want to know more about that.

For the assembly implementation I use 32 bit code in 16 bit mode of DOS... so you will see some mixing of 32 bit and 16 bit code... it is easy to convert it to complete 32 bit code. Note that the assembly part is fully tested to be working correctly, the Java or C code is derived from that.

Ok. Here is the assembly implementation for computing the CRC-32 table:

```
xor     ebx, ebx    ;ebx=0, because it will be used whole as pointer
InitTableLoop:
xor     eax, eax    ;eax=0 for new entry
mov     al, bl      ;lowest 8 bits of ebx are copied into lowest 8 bits of eax

;generate entry
xor     cx, cx
entryLoop:
```

```

        test    eax, 1
        jz      no_topbit
        shr     eax, 1
        xor     eax, poly
        jmp     entrygoon
no_topbit:
        shr     eax, 1
entrygoon:
        inc     cx
        test    cx, 8
        jz      entryLoop

        mov     dword ptr[ebx*4 + crctable], eax
        inc     bx
        test    bx, 256
        jz      InitTableLoop

```

Notes:

- crctable is an array of 256 dwords
- eax is shifted to the right because the CRC-32 uses reflected Algorithm
- also therefore the lowest 8 bits are processed...

In Java or C (int is 32 bit):

```

for (int bx=0; bx<256; bx++){
    int eax=0;
    eax=eax&0xFFFFF00+bx&0xFF;      // the 'mov al,bl' instruction
    for (int cx=0; cx<8; cx++){
        if (eax&&0x1) {
            eax>>=1;
            eax^=poly;
        }
        else eax>>=1;
    }
    crctable[bx]=eax;
}

```

The implementation for computing CRC-32 using the table:

```

computeLoop:
    xor     ebx, ebx
    xor     al, [si]
    mov     bl, al
    shr     eax, 8
    xor     eax, dword ptr[4*ebx+crctable]
    inc     si
    loop    computeLoop

    xor     eax, 0FFFFFFFh

```

Notes:

- ds:si points to the buffer where the bytes to process are
- cx contains the number of bytes to process
- eax contains current CRC
- crctable is the table computed with the code above
- the initial value of the CRC is in the case of CRC-32: FFFFFFFF
- after complete calculation the CRC is XORred with: FFFFFFFF which is the same as NOTting.

In Java or C it is like this:

```

for (int cx=0; cx>=8;
    eax^=crcTable[ebx];
}
eax^=0xFFFFFFFF;

```

So now we landed at the end of the first part: The CRC tutorial
 If you want to make a little deeper dive in CRC I suggest reading the document I did, you will find the URL at the end of this document.
 Ok. On to the most interesting part of this document: Reversing CRC!

Part 2: Reversing CRC

When I was thinking of a way to reverse it... I got stuck several times. I tried to 'deactivate' the CRC by thinking of such a sequence of bytes that it then shouldn't matter anymore what bytes you would place behind it. I couldn't do it... Then I realized it could NEVER work that way, because CRC algorithm is build in such a way it wouldn't matter which `_bit_` you would change, the complete CRC `_always_` (well always... almost) changes drastically. Try that yourself (with some simple CRC programs)... :)

I realized I only could 'correct' the CRC `_after_` the bytes I wanted to change. So I could make such a sequence of bytes, that would 'transform' the CRC into whatever I wanted!

Lets visualize the idea:

Bunch of bytes: 01234567890123456789012345678901234567890123456789012

You want to change from `^` this byte to `^` this one.

Thats position 9 to 26.

We also need 4 extra bytes (until position 30 `^`) for the sequence of bytes which will change the CRC back to its original value after the patched bytes.

When you are calculating the CRC-32 it goes fine until the byte on position 9, in the patched bunch of bytes the CRC radically changes from that point on. Even when pass position 26, from where the bytes are not changed, you never get the original CRC back. NOT! When you read the rest of this essay you know how. In short you have to do this when patching a certain bunch of bytes while maintaining the CRC:

1. Calculate the CRC until position 9, and save this value.
2. Continue calculating until position 27 and 4 extra bytes, save the resulting value.
3. Use the value of 1 for calculating the CRC of the 'new' bytes and the extra 4 bytes (this should be $27-9+4=22$ bytes) and save the resulting value.
4. Now we have the 'new' CRC value, but we want the CRC to be the 'old' CRC value. We use the reverse algorithm to compute the 4 extra bytes.

We can to point 1 to 3, below you learn to do point 4.

Reversing CRC-16

I thought, to make it more easy for you, first to calculate the reverse of CRC-16. Ok. We are on a certain point after the patched code where you want to change the CRC back to its original. We know the original CRC (calculated before patching the data) and the current CRC register. We want to calculate the 2-bytestring which changes the current CRC register to the original CRC. First we calculate 'normally' the CRC with the unknown 2 bytes naming them X and Y, for the register I take `a1 a0`, the only non-variable is zero (`00`). :) Look again at our latest CRC algorithm, figure 3, to understand better what im doing.

Ok, here we go:

Take a 2-bytestring 'X Y'. Bytes are processed from the left side.

Take for register `a1 a0`.

For a XOR operation I write '+' (as in the CRC tutorial)

Processing first byte, X:

<code>a0+X</code>	this is the calculated topbyte (1)
<code>b1 b0</code>	sequence in table where the topbyte points at
<code>00 a1</code>	to right shifted register
<code>00+b1 a1+b0</code>	previous 2 lines XORred with eachother

Now the new register is: (b1) (a1+b0)

Processing second byte, Y:


```

(a1+b0)+Y      this is the calculated topbyte (2)
c1 c0          sequence in table where the topbyte points at
00 b1          to right shifted register
00+c1 b1+c0    previous 2 lines XORred with eachother

```

Now the final register is: (c1) (b1+c0)

I'll show it a little different way:

```

a0 + X      =(1)  points to b1 b0 in table
a1 + b0 + Y =(2)  points to c1 c0 in table
      b1 + c0=d0  new low byte of register
      c1=d1      new high byte of register
(1) (2)

```

Wow! Let this info work out on you for a while... :)

Don't be afraid, a real value example is coming soon.

What if you wanted the register to be some d1 d0 (the original CRC) and you know the value of the register before the transformation (so a1 a0)... what 2 bytes or what X and Y would you have to fed through the CRC calculation?

Ok. We will begin working from the back to the front. d0 must be b1+c0 and d1 must be c1... But how-the-hell, I hear you say, can you know the value of byte b1 and c0??? Shall I remember you about the Table? You can just lookup the value of the word C0 C1 in the Table because you know C1. Therefore you need to make a 'lookup' routine. If you found the value, be sure to remember the index to the value because that's the way to find the unknown topbytes e.g. (1)&(2)!

So now you found c1 c0, how to get b1 b0? If b1+c0=d0 then b1=d0+c0! Now you use the lookup routine to lookup the b1 b0 value too. Now we know everything to calculate X & Y ! Cool huh?

```

a1+b0+Y=(2) so Y=a1+b0+(2)
a0+X=(1)    so X=a0+(1)

```

Non-variable example for CRC-16

Lets look at an example with real values:

```

-register before: (a1=)DE (a0=)AD
-wanted register: (d1=)12 (d0=)34

```

Look up the entry beginning with 12 in the CRC-16 table in the appendix.

-This is entry 38h with value 12C0. Try to find another entry beginning with 12. You can't find another because we calculated each entry for each possible value of the topbyte and that's 256 values, remember!

Now we know (2)= 38, c1= 12 and c0= C0, so b1= C0+34=F4, now look up the entry of B1 beginning with F4.

-This is entry 4Fh with value F441.

Now we know (1)= 4F, b1= F4 and b0= 41. Now all needed values are known, to compute X and Y we do:

```

Y=a1+b0+(2)=DE+41+38=A7
X=a0+(1)    =AD+4F    =E2

```

Conclusion: to change the CRC-16 register from DEAD to 1234 we need the bytes E2 A7 (in that order).

You see, to reverse CRC you have to 'calculate' your way back, and remember the values along the way. When you are programming the lookup table in assembly, remember that intel saves values backwards in Little-Endian format. Now you probably understand how to reverse CRC-16.... now CRC-32

Reversing CRC-32

Now we had CRC-16, CRC-32 is just as easy (or as difficult). You now work with 4 bytes instead of 2. Keep looking and comparing this with the 16bit version from above.

Take a 4-bytestring X Y Z W , bytes are taken from the LEFT side
 Take for register a3 a2 a1 a0

Note that a3 is the most significant byte and a0 the least.

Processing first byte, X:

a0+X				this is the calculated topbyte (1)
b3	b2	b1	b0	sequence in table where the topbyte points at
00	a3	a2	a1	to right shifted register
00+b3	a3+b2	a2+b1	a1+b0	previous 2 lines XORred with eachother

Now the new register is: (b3) (a3+b2) (a2+b1) (a1+b0)

Processing second byte, Y:

(a1+b0)+Y				this is the calculated topbyte (2)
c3	c2	c1	c0	sequence in table where the topbyte points at
00	b3	a3+b2	a2+b1	to right shifted register
00+c3	b3+c2	a3+b2+c1	a2+b1+c0	previous 2 lines XORred with eachother

Now the new register is: (c3) (b3+c2) (a3+b2+c1) (a2+b1+c0)

Processing third byte, Z:

(a2+b1+c0)+Z				this is the calculated topbyte (3)
d3	d2	d1	d0	sequence in table where the topbyte points at
00	c3	b3+c2	a3+b2+c1	to right shifted register
00+d3	c3+d2	b3+c2+d1	a3+b2+c1+d0	previous 2 lines XORred with eachother

Now the new register is: (d3) (c3+d2) (b3+c2+d1) (a3+b2+c1+d0)

Processing fourth byte, W:

(a3+b2+c1+d0)+W				this is the calculated topbyte (4)
e3	e2	e1	e0	sequence in table where the topbyte points at
00	d3	c3+d2	b3+c2+d1	to right shifted register
00+e3	d3+e2	c3+d2+e1	b3+c2+d1+e0	previous 2 lines XORred with eachother

Now the final register is: (e3) (d3+e2) (c3+d2+e1) (b3+c2+d1+e0)

I'll show it a little different way:

a0 + X	= (1)	points to	b3 b2 b1 b0	in table
a1 + b0 + Y	= (2)	points to	c3 c2 c1 c0	in table
a2 + b1 + c0 + Z	= (3)	points to	d3 d2 d1 d0	in table
a3 + b2 + c1 + d0 + W	= (4)	points to	e3 e2 e1 e0	in table
b3 + c2 + d1 + e0	= f0			
c3 + d2 + e1	= f1			
d3 + e2	= f2			
e3	= f3			
(1) (2) (3) (4)				

(figure 4)

This is reversed in the same way as the 16bit version. I shall give an example with real values. For the table values use the CRC-32 table in the appendix.

Take for CRC register before, a3 a2 a1 a0 -> AB CD EF 66

Take for CRC register after, f3 f2 f1 f0 -> 56 33 14 78 (wanted value)

Here we go:

First byte of entries	entry	value
e3=f3	=56 -> 35h=(4)	56B3C423 for e3 e2 e1 e0
d3=f2+e2 =33+B3	=E6 -> 4Fh=(3)	E6635C01 for d3 d2 d1 d0
c3=f1+e1+d2 =14+C4+63	=B3 -> F8h=(2)	B3667A2E for c3 c2 c1 c0
b3=f0+e0+d1+c2=78+23+5C+66=61	-> DEh=(1)	616BFFD3 for b3 b2 b1 b0

Now we have all needed values, then

X=(1)+ a0= DE+66=B8

Y=(2)+ b0+a1= F8+D3+EF=C4

Z=(3)+ c0+b1+a2= 4F+2E+FF+CD=53

W=(4)+d0+c1+b2+a3=35+01+7A+6B+AB=8E

(final computation)

Conclusion: to change the CRC-32 register from ABCDEF66 to 56331478 we need this sequence of bytes: B8 C4 53 8E

The reverse Algorithm for CRC-32

If you look at the by-hand computation of the sequence of bytes needed to change the CRC register from a3 a2 a1 a0 to f3 f2 f1 f0 its difficult to transform this into a nice compact algorithm.

Look at an extended version of the final computation:

	Position
X =(1) + a0	0
Y =(2) + b0 + a1	1
Z =(3) + c0 + b1 + a2	2
W =(4) + d0 + c1 + b2 + a3	3
f0= e0 + d1 + c2 + b3	4
f1= e1 + d2 + c3	5
f2= e2 + d3	6
f3= e3	7

(figure 5)

It is just the same as figure 4, only some values/bytes exchanged. This view will help us to get a compact algorithm. What if we take a buffer of 8 bytes that is, for every line you see in figure 5 one byte is reserved. Bytes 0 to 3 are filled with a0 to a3, bytes 4 to 7 are filled with f0 to f3. As before, we take the last byte e3 which is equal to f3 and lookup the complete value in the CRC table. Then we XOR this value (e3 e2 e1 e0) on position 4 (as in figure 5). Then we automatically know what the value of d3 is, because we already XORred f3 f2 f1 f0 with e3 e2 e1 e0, and f2+e2=d3. Because we now already know what the value of (4) is (the entry number), we can directly XOR the value into position 3. Now we know d3 use this to lookup the value of d3 d2 d1 d0 and XOR this on one position earlier, that is position 3 (look at the figure!). XOR the found entry number (3) for the value on position 2. We now know c3 because we have the value f1+e1+d2=c3 on position 5.

We go on doing this until we XORred b3 b2 b1 b0 on position 1. Et voila! Bytes 0 to 3 of the buffer now contains the needed bytes X to W!

Summarized is here the algorithm:

1. Of the 8 byte buffer, fill position 0 to 3 with a0 to a3 (the start value of the CRC register), and position 4 to 7 with f0 to f3 (wanted end value of CRC register).
2. Take the byte from position 7 and use it to lookup the complete value.
3. XOR this value (dword) on position 4
4. XOR the entry number (byte) on position 3
5. Repeat step 2 & 3 three more times while decreasing the positions each time by one.

Implementation of the Reverse Algorithm

Now its time for some code. Below are the implementation of the reverse algorithm for CRC-32 in Assembly (it is not difficult to do this for other languages and/or CRC standards). Note that in assembly (on PC's) dwords are written to and read from memory in reverse order.

```
crcBefore    dd (?)
wantedCrc    dd (?)
buffer       db 8 dup (?)

        mov     eax, dword ptr[crcBefore] ;/*
        mov     dword ptr[buffer], eax
        mov     eax, dword ptr[wantedCrc] ; Step 1
        mov     dword ptr[buffer+4], eax ;*/

        mov     di, 4
computeReverseLoop:
        mov     al, byte ptr[buffer+di+3] ;/*
        call    GetTableEntry             ; Step 2 */
        xor     dword ptr[buffer+di], eax ; Step 3
        xor     byte ptr[buffer+di-1], bl ; Step 4
```

```

    dec     di                ;/*
    jnz     computeReverseLoop ; Step 5 */

```

Notes:

-Registers eax, di bx are used

Implementation of GetTableEntry

```

crctable      dd 256 dup (?)      ;should be defined globally somewhere & initialized of course

    mov     bx, offset crctable-1
getTableEntryLoop:
    add     bx, 4                ;points to (crctable-1)+k*4 (k:1..256)
    cmp     [bx], al            ;must always find the value somewhere
    jne     getTableEntryLoop

    sub     bx, 3
    mov     eax, [bx]
    sub     bx, offset crctable
    shr     bx, 2

    ret

```

On return eax contains a table entry, bx contains the entry number.

Outtro

Well... you reached the end of this essay. If you now think: wow, all those programs which are protected by CRC can say 'bye, bye'. Nope. It is very easy to make an anti-anti-CRC code. To make a succesfull CRCreverse you have to know exactly from what part of the code the CRC is calculated and what CRC algorithm is used. A simple countermeasure is using 2 different CRC algorithms, or combination with another dataprotection algorithm.

Anywayz... I hope all this stuff was interesting and that you enjoyed reading it as I enjoyed writing it.

Fnx go out to the beta-testers Douby/DREAD and Knotty Dread for the good comments on my work which made it even better!

For a sample CRC-32 correcting patcher program visit my webpages:

<http://surf.to/anarchriz> -> Programming -> Projects
(it's still a preview but will give you a proof of my idea)

For more info on DREAD visit <http://dread99.cjb.net>

If you still have questions you can mail me at anarchriz@hotmail.com, or try the channels #DreaD, #Win32asm, #C.I.A and #Cracking4Newbies (in that order) on EFnet (on IRC).

CYA ALL! - Anarchriz

"The system makes its morons, then despises them for their ineptitude, and rewards its 'gifted few' for their rarity." - Colin Ward

Appendix

CRC-16 Table

00h	0000	C0C1	C181	0140	C301	03C0	0280	C241
08h	C601	06C0	0780	C741	0500	C5C1	C481	0440
10h	CC01	0CC0	0D80	CD41	0F00	CFC1	CE81	0E40
18h	0A00	CAC1	CB81	0B40	C901	09C0	0880	C841

20h	D801	18C0	1980	D941	1B00	DBC1	DA81	1A40
28h	1E00	DEC1	DF81	1F40	DD01	1DC0	1C80	DC41
30h	1400	D4C1	D581	1540	D701	17C0	1680	D641
38h	D201	12C0	1380	D341	1100	D1C1	D081	1040
40h	F001	30C0	3180	F141	3300	F3C1	F281	3240
48h	3600	F6C1	F781	3740	F501	35C0	3480	F441
50h	3C00	FCC1	FD81	3D40	FF01	3FC0	3E80	FE41
58h	FA01	3AC0	3B80	FB41	3900	F9C1	F881	3840
60h	2800	E8C1	E981	2940	EB01	2BC0	2A80	EA41
68h	EE01	2EC0	2F80	EF41	2D00	EDC1	EC81	2C40
70h	E401	24C0	2580	E541	2700	E7C1	E681	2640
78h	2200	E2C1	E381	2340	E101	21C0	2080	E041
80h	A001	60C0	6180	A141	6300	A3C1	A281	6240
88h	6600	A6C1	A781	6740	A501	65C0	6480	A441
90h	6C00	ACC1	AD81	6D40	AF01	6FC0	6E80	AE41
98h	AA01	6AC0	6B80	AB41	6900	A9C1	A881	6840
A0h	7800	B8C1	B981	7940	BB01	7BC0	7A80	BA41
A8h	BE01	7EC0	7F80	BF41	7D00	BDC1	BC81	7C40
B0h	B401	74C0	7580	B541	7700	B7C1	B681	7640
B8h	7200	B2C1	B381	7340	B101	71C0	7080	B041
C0h	5000	90C1	9181	5140	9301	53C0	5280	9241
C8h	9601	56C0	5780	9741	5500	95C1	9481	5440
D0h	9C01	5CC0	5D80	9D41	5F00	9FC1	9E81	5E40
D8h	5A00	9AC1	9B81	5B40	9901	59C0	5880	9841
E0h	8801	48C0	4980	8941	4B00	8BC1	8A81	4A40
E8h	4E00	8EC1	8F81	4F40	8D01	4DC0	4C80	8C41
F0h	4400	84C1	8581	4540	8701	47C0	4680	8641
F8h	8201	42C0	4380	8341	4100	81C1	8081	4040

CRC-32 Table

00h	00000000	77073096	EE0E612C	990951BA
04h	076DC419	706AF48F	E963A535	9E6495A3
08h	0EDB8832	79DCB8A4	E0D5E91E	97D2D988
0Ch	09B64C2B	7EB17CBD	E7B82D07	90BF1D91
10h	1DB71064	6AB020F2	F3B97148	84BE41DE
14h	1ADAD47D	6DDDE4EB	F4D4B551	83D385C7
18h	136C9856	646BA8C0	FD62F97A	8A65C9EC
1Ch	14015C4F	63066CD9	FA0F3D63	8D080DF5
20h	3B6E20C8	4C69105E	D56041E4	A2677172
24h	3C03E4D1	4B04D447	D20D85FD	A50AB56B
28h	35B5A8FA	42B2986C	DBBBC9D6	ACBCF940
2Ch	32D86CE3	45DF5C75	DCD60DCF	ABD13D59
30h	26D930AC	51DE003A	C8D75180	BFD06116
34h	21B4F4B5	56B3C423	CFBA9599	B8BDA50F
38h	2802B89E	5F058808	C60CD9B2	B10BE924
3Ch	2F6F7C87	58684C11	C1611DAB	B6662D3D
40h	76DC4190	01DB7106	98D220BC	EFD5102A
44h	71B18589	06B6B51F	9FBFE4A5	E8B8D433
48h	7807C9A2	0F00F934	9609A88E	E10E9818
4Ch	7F6A0DBB	086D3D2D	91646C97	E6635C01
50h	6B6B51F4	1C6C6162	856530D8	F262004E
54h	6C0695ED	1B01A57B	8208F4C1	F50FC457
58h	65B0D9C6	12B7E950	8BBEB8EA	FCB9887C
5Ch	62DD1DDF	15DA2D49	8CD37CF3	FBD44C65

60h	4DB26158	3AB551CE	A3BC0074	D4BB30E2
64h	4ADFA541	3DD895D7	A4D1C46D	D3D6F4FB
68h	4369E96A	346ED9FC	AD678846	DA60B8D0
6Ch	44042D73	33031DE5	AA0A4C5F	DD0D7CC9
70h	5005713C	270241AA	BE0B1010	C90C2086
74h	5768B525	206F85B3	B966D409	CE61E49F
78h	5EDEF90E	29D9C998	B0D09822	C7D7A8B4
7Ch	59B33D17	2EB40D81	B7BD5C3B	C0BA6CAD
80h	EDB88320	9ABFB3B6	03B6E20C	74B1D29A
84h	EAD54739	9DD277AF	04DB2615	73DC1683
88h	E3630B12	94643B84	0D6D6A3E	7A6A5AA8
8Ch	E40ECF0B	9309FF9D	0A00AE27	7D079EB1
90h	F00F9344	8708A3D2	1E01F268	6906C2FE
94h	F762575D	806567CB	196C3671	6E6B06E7
98h	FED41B76	89D32BE0	10DA7A5A	67DD4ACC
9Ch	F9B9DF6F	8EBEEFF9	17B7BE43	60B08ED5
A0h	D6D6A3E8	A1D1937E	38D8C2C4	4FDF252
A4h	D1BB67F1	A6BC5767	3FB506DD	48B2364B
A8h	D80D2BDA	AF0A1B4C	36034AF6	41047A60
ACh	DF60EFC3	A867DF55	316E8EEF	4669BE79
B0h	CB61B38C	BC66831A	256FD2A0	5268E236
B4h	CC0C7795	BB0B4703	220216B9	5505262F
B8h	C5BA3BBE	B2BD0B28	2BB45A92	5CB36A04
BCh	C2D7FFA7	B5D0CF31	2CD99E8B	5BDEAE1D
C0h	9B64C2B0	EC63F226	756AA39C	026D930A
C4h	9C0906A9	EB0E363F	72076785	05005713
C8h	95BF4A82	E2B87A14	7BB12BAE	0CB61B38
CCh	92D28E9B	E5D5BE0D	7CDCEFB7	0BDBDF21
D0h	86D3D2D4	F1D4E242	68DDB3F8	1FDA836E
D4h	81BE16CD	F6B9265B	6FB077E1	18B74777
D8h	88085AE6	FF0F6A70	66063BCA	11010B5C
DCh	8F659EFF	F862AE69	616BFFD3	166CCF45
E0h	A00AE278	D70DD2EE	4E048354	3903B3C2
E4h	A7672661	D06016F7	4969474D	3E6E77DB
E8h	AED16A4A	D9D65ADC	40DF0B66	37D83BF0
ECh	A9BCAE53	DEBB9EC5	47B2CF7F	30B5FFE9
F0h	BDBDF21C	CABAC28A	53B39330	24B4A3A6
F4h	BAD03605	CDD70693	54DE5729	23D967BF
F8h	B3667A2E	C4614AB8	5D681B02	2A6F2B94
FCh	B40BBE37	C30C8EA1	5A05DF1B	2D02EF8D

References

- > A painless guide to CRC error detection algorithm
url: ftp://ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt
(I bet this 'painless guide' is more painfull then my 'short' one ;)
- > I also used a random source of a CRC-32 algorithm to understand the algorithm better.
- > Link to crc calculation progs... hmmm search for 'CRC.ZIP' or 'CRC.EXE' or something alike at ftpsearch (http://ftpsearch.lycos.com?form=advanced)

Copyright (c) 1998,1999 by Anarchriz
(this is REALLY the last line :)