Humboldt University Berlin
Computer Science Department
Systems Architecture Group

Rudower Chaussee 25
D-12489 Berlin-Adlershof
Germany

Phone: +49 30 2093-3400
Fax: +40 30 2093-3112
http://sar.informatik.hu-berlin.de

Reversing CRC – Theory and Practice.

HU Berlin Public Report
SAR-PR-2006-05

May 2006

Authors:
Martin Stigge, Henryk Plötz, Wolf Müller, Jens-Peter Redlich

Reversing CRC – Theory and Practice

Martin Stigge, Henryk Plötz, Wolf Müller, Jens-Peter Redlich
(mstigge|ploetz|wolfm|jpr)@informatik.hu-berlin.de
Systems Architecture Group*, Computer Science Department,
Humboldt-University Berlin, Unter den Linden 6, 10099 Berlin, Germany

24th May 2006

Abstract

The Cyclic Redundancy Check (CRC) was developed as a checksum algorithm for the detection of data corruption in the process of data transmission or storage. However, in some scenarios there's a CRC given which a set of data is expected to have, so the data itself has to be modified (at the end or at some chosen position) in a way that it computes to the given CRC checksum afterwards. We present methods providing solutions to this problem. Each algorithm is explained in theory and accompanied by an implementation for the CRC32 in the C programming language.

1 Introduction

The process of data transmission or storage usually contains the risk of unwanted modification of the data at the most physical level, caused by noisy or damaged transmission or storage media. (This does not include alteration by an intelligent third party like a malicious attacker.) To detect these errors, some error-detecting and even -correcting codes were invented, which calculate a value from the set of data and transmit or store it with the data. Any hash function can be used to perform this kind of error detection to a certain degree, and one of them is the "Cyclic Redundancy Check" (CRC). It's not a cryptographically secure hash and therefore can not reliably detect malicious changes in the transmitted data, but it can provably detect some common accidental errors like single-/two-bit or burst errors and can additionally be implemented very efficiently. There are different instances of the CRC which mainly differ in the polynomial on which they are based on, resulting in different sizes of the computed value. The most popular one is the CRC32, which computes a 32-bit value.[1]

While most of the time you want to calculate the CRC of a given set of data, there are some situations where the CRC is given and you want to modify your data so that it computes to this CRC value afterwards. These scenarios include hard-wired checksums of firmware or calculating the CRC of a set of data which

*http://sar.informatik.hu-berlin.de

[1] There are different instances of CRC out there with a width of 32 bits. With "CRC32" we always mean the CRC used within IEEE 802 and many other standards, which is different from e.g. CRC-32/Castagnoli.

includes the CRC itself. An example of the latter case is the creation of a ZIP archive which includes itself as a file, see [PSMR06] for details. We develop and analyse methods to calculate these modifications within the next sections.

This article is structured in the following way: Section 2 gives an overview on how the CRC is mathematically defined and how it is calculated in practice. This is meant as an overview as only those aspects being important to understand our "reversal" of the CRC are explained in detail. Section 3 then deals with the question of how to manipulate the end of your data so that you know in advance which CRC will be calculated, regardless of your actual data. After that, section 4 will explain how to do the same with the ability to not only know the CRC in advance but even to choose an arbitrary value for the CRC. This is extended in section 5 where we develop a method to do this manipulation anywhere we want within our data, possibly far away from the end. (Its subsection 5.3 describes the most flexible and elegant solution for this, so you can jump directly there if you are just looking for the required steps.) We finally draw some conclusions in section 6 before our implementations of all algorithms are presented in the appendix A.

Within the sections, each algorithm is first developed and explained in theory. Second, this is summarized by presenting some pseudo-code, which is easy to read and corresponds directly to the other theoretical background. This pseudo-code will also be independent of the CRC instance used and accompanied by an example for better understanding. Third, you'll find working and well-tested C code for each of the algorithms collected in the appendix A. These will be implemented for use with the well-known CRC32 standard.

2 How CRC(32) Works

The CRC itself is essentially one giant polynomial division which can be efficiently implemented in software and hardware (i.e. in $O(n)$ time). There are many publications dealing with this topic in a very detailed way (see e.g. [Wil96] or [Tan81]), so we will describe the concept only to a degree needed to understand the details of our "reversal" methods.

We will look at the CRC in 3 more or less different ways: The "algebraic approach", the "bit-oriented approach" and the "table-driven approach". The algebraic approach is the way the CRC is mathematically defined and is not much more than the polynomial division mentioned above, but it's often far too much maths for those who do the practical work, i.e. write the code. That's why we'll also look at the bit-oriented approach which is a polynomial division in practice by operating directly on the bits of the input data. Finally, there is the table-driven approach, which does the same work in a faster and more efficient way, which is why this is the way real-world programs actually compute the CRC.

Before presenting the general ideas of the three approaches, some words about the bit-ordering: The literature about this topic sometimes uses the concept of "reflection" which is about the ordering of the calculated bits. It does not really make any difference which ordering you use, as long as you do this in a consistent way. We will avoid to look at both types of bit-ordering within this article: We always start counting the data from bit 0, from "the left to the right" and do the same with the bits of the used CRC register and the CRC

polynomial. This will avoid confusion and make everything more consistent. The important thing to note is that bytes are usually noted writing the least significant bit (with number 0) to the right. To keep consistency, we will retain the numbering (instead of the "left to right" ordering), for instance leading to a "right shift" (>>) within the C code where we and the figures make a "left shift" (meaning: shifting in direction of lower indices) and vice versa.

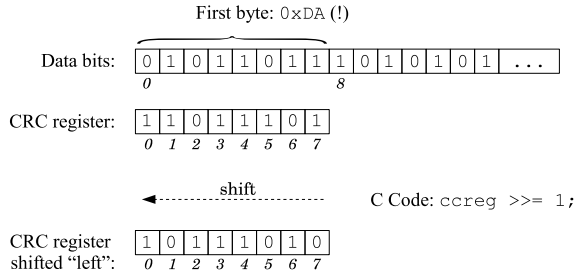

Figure 1: Illustration of the bit-ordering we use

And now for the details of the three approaches: (Again, please look at the aforementioned publications for more details.)

Algebraic approach: By definition, the CRC is more or less one gigantic polynomial division: The data is interpreted as the coefficients of a giant polynomial which is divided by a given CRC polynomial. The remainder of this division is the CRC.

What kind of polynomial do we have for our data? All the data-bits are interpreted as elements from $\mathbb{F}_2 = \{0, 1\}$ where we can define an addition $\oplus$ (which is essentially XOR) and multiplication $\odot$ (which is AND). The set of polynomials is therefore called $\mathbb{F}_2[x]$, and a polynomial for data-bits $a = a_0 \dots a_{l-1}$ looks like this:

$$a(x) = a_0 x^{l-1} + a_1 x^{l-2} + \ldots + a_{l-2} x + a_{l-1}$$

Addition within this set of polynomials is invertible (with every $p(x) \in \mathbb{F}_2[x]$ being its own inverse: $p(x) = -p(x)$), but unfortunately, the multiplication is not. So the division of two polynomials may give a remainder. Having that in mind, the polynomial division of a polynomial $p(x)$ by another polynomial $q(x)$ can be expressed as finding $s(x)$ so that there is $r(x)$ (the remainder polynomial) with a degree of less than the degree of $q(x)$ so that:

$$p(x) = s(x) \cdot q(x) + r(x)$$

Using this, the set of polynomial congruence classes $\mathbb{F}_2[x]/p_{\mathrm{CRC}}(x)$ can be defined: Each element $r(x)$ within this set is one of the possible remainders and represents all polynomials which leave $r(x)$ as the remainder when devided by $p_{\mathrm{CRC}}(x)$. The computation of the CRC is a polynomial division which computes the remainder of our data-polynom (after it's multiplied by x^N for technical reasons). So finally, calculating the CRC is defined as finding a polynomial $b(x)$ so that there is an $r(x)$ with a degree of less than N so that:

$$a(x) \cdot x^N = b(x) \cdot p_{\mathrm{CRC}}(x) + r(x)$$

Please note that we are not interested in how $b(x)$ looks like.

Bit-oriented Algorithm: A very simple and naive approach to implement this is to simply see the bit-stream of input data ("augmented" with N 0-bits to have all coefficients of $a(x) \cdot x^N$) and add (=subtract) the coefficients of $p_{\mathrm{CRC}}(x)$ where needed. You do this until the remaining bits "span" a shorter range than N, so they are the remainder coefficients and therefore the CRC we looked for. This works because we are not interested in the quotient but only the remainder of the division.

Practically, we could use a bit-register with a width of N where the data-bits are "shifted in" from the right and as soon as a 1 is shifted out of the left side, everything is xored with the coefficients of the CRC polynomial $p_{\mathrm{CRC}}(x)$. The result would be the same. The bit-mask which is xored is called the CRCPOLY. This approach is improved a bit: The register (from now on called the CRC register) will in practice store only the effect of the xor operations on the data stream, not the result itself. This means that we start with a register of all zeros and will shift in 0 from the right. We will xor the CRCPOLY when the bit just shifted out is different from the bit we see in the data stream. This has the added benefit that the "augmentation" with N 0-bits is not needed anymore, because these bits are never considered in any operation.

For technical reasons, the whole process doesn't start with a CRC register of all 0, but with a pattern which is meant to compensate for errors like erroneously added or left-out leading zeros which would otherwise remain undetected. We will call this pattern the INITXOR and its usually all 1. Symmetrically, there is a second pattern we call FINALXOR which is added to the CRC register after the computation. We have to keep this in mind, as soon as the result will be used as a CRC value, but it doesn't change the structure of the algorithm itself.

To summarize this algorithm, look at the pseudo-code in algorithm 1, which will calculate the CRC in the manner above. Its implementation in C looks quite similar and can be found in the appendix A.2.

Algorithm 1 Bit-oriented calculation of the CRC

Input: a (containing the data bits)
  $crcreg \leftarrow$ INITXOR
  for $i = 0$ to $l - 1$ **do**
    LeftShift($crcreg$)
    if $bit\_just\_shifted\_out \neq a_i$ **then**
      $crcreg \leftarrow crcreg \oplus$ CRCPOLY
    end if
  end for
  $crcreg \leftarrow crcreg \oplus$ FINALXOR
Output: $crcreg$

Table-driven Algorithm: The bit-oriented approach is not very efficient as it operates at the bit-level resulting in one loop for each bit, regardless of the word-width which your machine supports. To improve this, it's possible to process units of multiple bits at once. The idea is to shift not only one bit at a time but M instead. What we get is a pattern of M bits shifted out of the

register which have to be compared to corresponding M bits of the data stream (which we divide into blocks of M bits). After that, we have to apply a pattern (xor-mask) to the M bits just shifted out and the N bits of the CRC register so that the first M bits match the M bits of the data stream. This xor-mask (of $M + N$ bits) has to be the sum of correctly shifted CRCPOLYs so that it will make the M bits shifted out equal to the M bits of the data stream. (In practice it only has be to applied to the N bits of the CRC register because the first M bits will afterwards be equal to the data bits and therefore discarded.)

The process of finding such a sum of CRCPOLYs is equivalent to calculating the bit-oriented approach – but it only has to be done once in advance. For every pattern of M bits shifted out and xored with the data bits (so that's the pattern we have to xor actually), the corresponding mask of N bits which has to be applied to the CRC register can be stored in a table. This table is called the "CRC table". It has 2^M lines with N bits each. The typical size for CRC32 ($N = 32$) is $M = 8$ so that the units to be processed are bytes and the table has $2^8 = 256$ entries (and thus a size of $2^8 \cdot 32$ bits which is 1 kilobyte). With $M = 16$ the process of calculating the CRC32 would be twice as fast, but the table would be 2^8 times larger (256 kilobytes) having an index of 2^{16}.

Again, this is only a short introduction to the algorithms. Please look at both algorithms in pseudo-code in algorithms 2 and 3, which will calculate the CRC table and the CRC in the above manner. Their implementations in C can also be found in the appendix A.3.

Algorithm 2 Calculating the CRC table

Input: (nothing)
  for $index = 0$ to $2^M - 1$ **do**
    $crcreg \leftarrow 0$ {Note that $crcreg$ is N bits width!}
    $crcreg_0 \cdots crcreg_{M-1} \leftarrow index_0 \cdots index_{M-1}$
    for $k = 1$ to M **do**
      LeftShift($crcreg$)
      if $bit\_just\_shifted\_out = 1$ **then**
        $crcreg \leftarrow crcreg \oplus$ CRCPOLY
      end if
    end for
    $crctable[index] \leftarrow crcreg$
  end for
Output: $crctable$

2.1 Notation

After having seen how the CRC register works, we will introduce some notation that is used in the following sections. This deals with the hexadecimal notation of polynomials as well as some (mathematical) functions operating on words of bits which we define here for later use (and which are mostly related to the bit-oriented view).

- The representation of polynomials is often given in a hexadecimal notation, where the bits represent the coefficients of the polynomial. It's *really*

Algorithm 3 CRC() – Table-driven calculation of the CRC

Input: a (containing the data bits)
  $crcreg \leftarrow$ INITXOR
  for $i = 0$ to $(l/M) - 1$ **do**
    LEFTSHIFT($crcreg$, M)
    $index \leftarrow bits\_just\_shifted\_out \oplus a_i \cdots a_{i+M-1}$
    $crcreg \leftarrow crcreg \oplus crctable[index]$
  end for
  $crcreg \leftarrow crcreg \oplus$ FINALXOR
Output: $crcreg$

important to note, that they are in a "reverse order", because the polynomial "starts" with the coefficient of index 0 which will be the bit of index 0 and therefore the least significant bit. This means that e.g. a polynomial $p(x) = x^2 + 1$ will be represented as `0xA`.

- The CRC polynomial itself which we called $p_{\mathrm{CRC}}(x)$ is represented in CRCPOLY, but as described in the above subsection this is a special case, because the highest coefficient is omitted for practical reasons (which is no problem because it's always 1 and its degree is N). This means that e.g. CRCPOLY = `0x94` with $N = 8$ would be $p_{\mathrm{CRC}}(x) = x^8 + x^5 + x^3 + 1$.

- Now for the functions: With $\mathrm{crc}(a)$ we denote the function which computes just the remainder of the polynomial division itself, where a contains the coefficients of the polynomial to be divided. Implicitly, the divisor polynomial is always $p_{\mathrm{CRC}}(x)$ if not stated otherwise.

- We "overload" the function crc with a different signature: $\mathrm{crc}(r, a)$ is the function which computes the remainder but starts with a CRC register of r. This will be used when starting a computation somewhere within the data where some computation has already been done and the CRC register has already some value different from the initial value. Thus, $\mathrm{crc}(a) = \mathrm{crc}(0, a)$.

- Further, we use $\mathrm{CRC}(a)$ for the function that applies the INITXOR and FINALXOR to the CRC register before and after the computation. This is the real-world function of computing such a checksum. Using $\mathrm{crc}(.,.)$, this can be written as: $\mathrm{CRC}(a) = \mathrm{crc}(\text{INITXOR}, a) \oplus \text{FINALXOR}$

- The function $\mathrm{CRC}\,32(a)$ is the special case of $\mathrm{CRC}(a)$ where we have implicitly the following values:
  - $N = 32$, and typical implementations use $M = 8$ (byte-blocks)
  - $p_{\mathrm{CRC}}(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
  - This is used as: CRCPOLY = `0xEDB88320`
  - INITXOR = FINALXOR = `0xFFFFFFFF`

3 How to get a known CRC

3.1 What to do

In many cases, you don't want to chose an arbitrary CRC value, you just want to have your data compute to a CRC which you know beforehand. It's very easy to change your data to compute to a crc of 0: Just append the old crc directly:

$$\mathrm{crc}(a_0 \cdots a_{l-1}\, \mathrm{crc}(a_0 \cdots a_{l-1})) = 0$$

If your data has a fixed size, this means that you compute the crc of your data with the exception of the last N bits and then replace those with the computed value.

Note that crc denotes the remainder of the polynomial division itself, which will not be the same as the final CRC value as soon as INITXOR and FINALXOR are not 0, so you possibly have to apply FINALXOR to the output of a CRC function (INITXOR does not matter in this case), and the result you will get is the FINALXOR instead of 0:

$$\mathrm{CRC}(a_0 \cdots a_{l-1}(\mathrm{CRC}(a_0 \cdots a_{l-1}) \oplus \mathrm{FINALXOR})) = \mathrm{FINALXOR}$$

For CRC 32 (where FINALXOR = `0xFFFFFFFF`) this means (note that $\overline{x}$ denotes the one's complement of x):

$$\mathrm{CRC}\,32(a_0 \cdots a_{l-1}\overline{\mathrm{CRC}\,32(a_0 \cdots a_{l-1})}) = \texttt{0xFFFFFFFF}$$

You may also omit (or forget) to apply FINALXOR before appending the CRC to your data, which may result in a crc unequal to 0 (and thus CRC unequal to FINALXOR), but which is surprisingly still independent of your actual data (with regard to the important precondition that INITXOR = FINALXOR):

$$\mathrm{CRC}(a_0 \cdots a_{l-1}\, \mathrm{CRC}(a_0 \cdots a_{l-1})) = m$$

We call it m, the "magic sequence", as it's magically the same value you get if you calculate the CRC of 0, i.e. $m := \mathrm{CRC}(0^N)$. For CRC 32 you'll find that $m =$ `0x2144DF1C`, so for the case of CRC 32 this means:

$$\mathrm{CRC}\,32(a_0 \cdots a_{l-1}\, \mathrm{CRC}\,32(a_0 \cdots a_{l-1})) = \texttt{0x2144DF1C}$$

Conclusively, the easiest way to get a known CRC 32 is to append the old CRC 32 to your data which will always give you a CRC 32 of `0x2144DF1C`. For other instances of CRC where INITXOR $\neq$ FINALXOR (e.g. CRC 16) the easiest way is to apply FINALXOR to the CRC before appending it, which will give you a CRC equal to FINALXOR. Figure 2 summarizes the process. We will not give any pseudo-code for that, because it's too simple. Please look at the appendix A.4 for a C implementation.

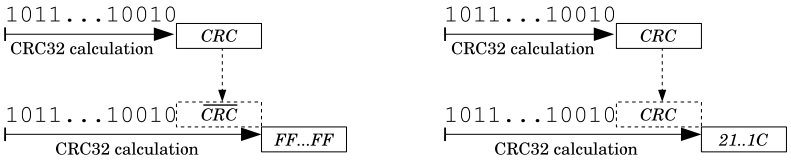

Figure 2: How to get a known CRC

3.2 Why this always works

First, we'll explain why the remainder is 0 if you append the old remainder to your data. After that, the mystery about m is revealed.

The case of CRC = FINALXOR

The first thing is easy to explain if we remind ourselves of the mathematics behind the computation. If your data is the sequence $a_0 \cdots a_{l-1}$ of coefficients for the polynomial $a(x)$ and $p_{\mathrm{CRC}}(x)$ is the divisor polynomial, we get the crc value as the sequence of N coefficients for the polynomial $r_{\mathrm{old}}(x)$ which was defined as the remainder of the polynomial division:

$$a(x) \cdot x^N = b(x) \cdot p_{\mathrm{CRC}}(x) + r_{\mathrm{old}}(x) \tag{1}$$

So what happens if we append the crc value to our data a to get the new data a'? The data a is "left-shifted", which is in math terms a multiplication of $a(x)$ with x^N followed by an addition of $r_{\mathrm{old}}(x)$, so our new polynomial looks like this:

$$\underbrace{\big(a(x) \cdot x^N + r_{\mathrm{old}}(x)\big)}_{=:a'(x)} \cdot x^N$$

Remember that the "data-polynomial" is always multiplied by x^N before the polynomial division takes place. Now we transform this expression using (1) to easily see which remainder this would give after the polynomial division:

$$\begin{aligned} a'(x) \cdot x^N &= \big(a(x) \cdot x^N + r_{\mathrm{old}}(x)\big) \cdot x^N \\ &\overset{(1)}{=} \big(b(x) \cdot p_{\mathrm{CRC}}(x) + \underbrace{r_{\mathrm{old}}(x) + r_{\mathrm{old}}(x)}_{=0}\big) \cdot x^N \\ &= b(x) \cdot x^N \cdot p_{\mathrm{CRC}}(x) + 0 \end{aligned}$$

(Note that $p(x) + p(x) = 0$ for every polynomial $p(x)$ in $\mathbb{F}_2[x]$, each is its own additive inverse.) As you see, the remainder $r_{\mathrm{new}}(x)$ would be 0, and that's why crc = 0 in this case.

This directly leads to CRC = FINALXOR. (Note that INITXOR does not matter because using INITXOR $\neq 0$ is equivalent to applying it to the first N bits of your data and instead using INITXOR = 0, this is explained in 2.)

If this already was to much math for you, just look at the pseudo-code given in section 2: For the last N runs, the inner loop will not apply the CRCPOLY because the bit shifted out of the CRC register (which is $r_{\mathrm{old}}(x)$ just before the last N runs) will be the same as the data bit (by definition, because we appended

the $r_{\text{old}}(x)$, i.e. contents of the CRC register). So the content of *crcreg* will finally be completely shifted out, resulting in a value of 0, see figure 3.

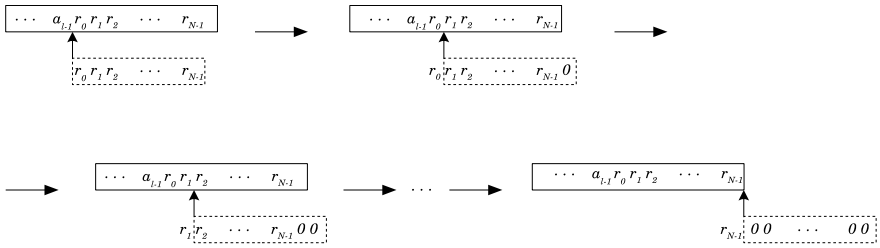

Figure 3: The *crcreg* is shifted into the 0 state.

The case of CRC = m

Secondly, we explore the property of m which we defined as $m := \text{CRC}(0^N)$. Remember (or see section 2) the function $\text{crc}(r', b) = r$ which means that if we start with a CRC register of r' and apply the CRC algorithm to the data bits of b, we get r as the new content of the CRC register. In these terms, $\text{CRC}(0^N) = m$ can be written as:

$$\text{crc}(\text{INITXOR}, 0^N) = m \oplus \text{FINALXOR} \tag{2}$$

We also know from above that appending the contents of the CRC register to the data always leads to crc = 0, and this can be expressed as:

$$\forall r \in \{0,1\}^N : \text{crc}(r, r) = 0 \tag{3}$$

If you add (2) and (3) you finally get:

$$\begin{aligned} m \oplus \text{FINALXOR} &= \text{crc}(\text{INITXOR}, 0^N) \oplus \text{crc}(r, r) \\ &\overset{(*)}{=} \text{crc}(\text{INITXOR} \oplus r, 0^N \oplus r) \\ &= \text{crc}(\text{FINALXOR} \oplus r, r) \end{aligned}$$

If we skip $(*)$ for a moment and accept the last equation because we required INITXOR = FINALXOR, then we finally get for $r := \text{CRC}(a)$:

$$\text{CRC}(a\,\text{CRC}(a)) = \underbrace{\text{crc}(\text{CRC}(a) \oplus \text{FINALXOR}, \text{CRC}(a))}_{=m \oplus \text{FINALXOR}} \oplus \text{FINALXOR} = m$$

This proves that appending the CRC of any data to this data, the resulting CRC will (regardless of the data itself) always be the same m which is characteristic for the used CRCPOLY, under the precondition that INITXOR = FINALXOR.

We skipped $(*)$, which holds because of the following property of crc():

$$\text{crc}(r_1, a_1) \oplus \text{crc}(r_2, a_2) = \text{crc}(r_1 \oplus r_2, a_1 \oplus a_2) \tag{4}$$

This kind of additive homomorphism can be easily explained by looking at the bit-oriented approach, see figure 4.

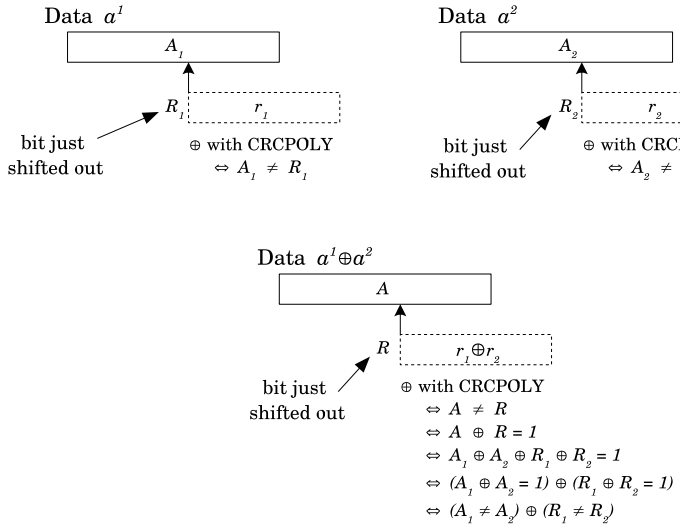

Figure 4: Additive homomorphism of crc(.,.)

Let's assume that we have a situation where the CRC register contains $r_1 \oplus r_2$ and we read a bit from $a_1 \oplus a_2$. The CRCPOLY is xored if and only if *either* it would be xored with r_1 in the CRC register while reading from a_1 *or* with r_2 in the CRC register while reading from a_2. Therefore, the new contents of the CRC register is exactly the XOR of the other two instances, as it was before. Inductively, the contents of the CRC register is still the $\oplus$ of both instances after processing all of $a_1 \oplus a_2$. (We will use this property again in section 5 to develop an extremely efficient way for altering a chosen position to get a chosen CRC.)

4 How to get a chosen CRC

4.1 Theory behind it

The first thing that came to my mind was to simply reverse the bit-oriented algorithm, and an implementation of this worked fine. Practically, this means to just look for the positions where to xor the CRCPOLY to the CRC register and to adjust the needed input-bits accordingly. There are other approaches (e.g. at [ana99], [Wes05] or even [Wes03]) which do essentially the same thing for the table-driven algorithm by fiddling around with the entries in the pre-built CRC table (see section 2). While those approaches basically work, there's a much clearer and simpler solution if you look at the math from which this is derived, i.e. by reversing the algebraic approach.

Let's assume that we have some data $a = a_0 \cdots a_{l-1}$ which leaves a remainder of $r_{\text{old}} = r_0 \cdots r_{N-1}$. Using the function crc, this can be denoted as:

$$\text{crc}(a_0 \cdots a_{l-1}) = r_0 \cdots r_{N-1}$$

As we know, this is essentially a polynomial division, which can be written as:

$$a(x) \cdot x^N = b(x) \cdot p_{\mathrm{CRC}}(x) + r_{\mathrm{old}}(x)$$

Now let's extend a by $\tilde{a} = \tilde{a}_0 \cdots \tilde{a}_{N-1}$ so that the new remainder is a chosen $r_{\mathrm{new}} = r'_0 \cdots r'_{N-1}$. This is written as:

$$\mathrm{crc}(a_0 \cdots a_{l-1}\tilde{a}_0 \cdots \tilde{a}_{N-1}) = r'_0 \cdots r'_{N-1}$$

What happens to the polynomials? Similar to section 3.2 we find the following:

$$\begin{aligned}\big(a(x) \cdot x^N + \tilde{a}(x)\big) \cdot x^N &= (b(x) \cdot p_{\mathrm{CRC}}(x) + r_{\mathrm{old}}(x) + \tilde{a}(x)) \cdot x^N \\ &= b(x) \cdot x^N \cdot p_{\mathrm{CRC}}(x) + (r_{\mathrm{old}}(x) + \tilde{a}(x)) \cdot x^N \\ &\equiv (r_{\mathrm{old}}(x) + \tilde{a}(x)) \cdot x^N\end{aligned}$$

(Note that we use the symbol $\equiv$ to denote the same remainder when divided by $p_{\mathrm{CRC}}(x)$.) We want this to be the new remainder $r_{\mathrm{new}}(x)$:

$$r_{\mathrm{new}}(x) \equiv (r_{\mathrm{old}}(x) + \tilde{a}(x)) \cdot x^N$$

We are looking for the coefficients of $\tilde{a}(x)$, and under a certain precondition (given below) we find them easily:

$$\tilde{a}(x) \equiv r_{\mathrm{new}}(x) \cdot (x^N)^{-1} + r_{\mathrm{old}}(x)$$

(Note that section 3.2 deals with a special case of this, where $r_{\mathrm{new}}(x) = 0$ so that $\tilde{a}(x) = r_{\mathrm{old}}(x)$ which was exactly what we found there. Note further, that the case of $r_{\mathrm{new}} = m \oplus \mathrm{FINALXOR}$ is also just a special case where $\tilde{a} = r_{\mathrm{old}} \oplus \mathrm{FINALXOR}$, so all the "magic" of m is hidden in the property $m(x) \equiv \big(x^N + 1\big) \cdot \mathrm{FINALXOR}(x)$.)

The precondition mentioned above is that x^N is invertible within the ring of polynomial congruence classes (which means that there is a $q(x)$ so that $x^N \cdot q(x) \equiv 1 \mod p_{\mathrm{CRC}}(x)$, which makes $q(x)$ the multiplicative inverse of x^N, and hence it's also called $(x^N)^{-1}$). Luckily, in the case of CRC32 the polynomial $p_{\mathrm{CRC}}(x)$ is irreducible, so this is even a field (it's isomorphic to the $\mathbb{F}_{2^8}$ widely used in cryptology) where every polynomial $p(x) \neq 0$ is invertible. But also in other cases of CRC polynomials, x^N has an inverse, as long as the coefficient of x^0 within $p_{\mathrm{CRC}}(x)$ is 1 (and for structural reasons, all used CRC polynomials have this property), because the only prime divisor x of x^N doesn't divide the CRC polynomial in these cases, making $p_{\mathrm{CRC}}(x)$ and x^N coprime so that $(x^N)^{-1}$ exists.

Conclusively, to find the coefficients of $\tilde{a}(x)$ which are the bits to be appended, we just have to multiply the wanted remainder with the inverse of x^N and finally add the old remainder. The coefficients of $(x^N)^{-1}$ can be precalculated (using the extended Euclidean algorithm or simply your favourite algebra program which implements it) as they only depend on $p_{\mathrm{CRC}}(x)$. For CRC32, the inverse of x^N is the following:

$$\begin{aligned}(x^{32})^{-1} \equiv\ & x^{31} + x^{30} + x^{27} + x^{25} + x^{24} + x^{23} + x^{22} + x^{21} + x^{20} + x^{16} \\ & + x^{15} + x^{13} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x^3 + x^1\end{aligned}$$

In this case, $(x^N)^{-1}$ can be expressed as CRCINV = `0x5B358FD3` which we'll use within our code. The multiplication (within the ring of polynomial congruence classes) can be implemented very efficiently as well as the final addition.

4.2 Pseudo-code

This pseudo-code summarizes our algebraic-motivated approach. Note that the multiplication itself is not "implemented" in detail here. This can be done by iterating over all bits of one operand and add the other operand to the result accordingly, while shifting it each time and reducing (i.e. subtracting the modulus) if necessary. Refer to our C implementation in the appendix A.5 for a working version of this.

Algorithm 4 AdjustData() – Data adjustment at the end by multiplication of r_{new} with $\left(x^N\right)^{-1}$ and adding r_{old}

Input: a (containing the data bits), $tcrcreg$ (the wanted CRC)
$tcrcreg \leftarrow tcrcreg \oplus \text{FINALXOR}$
$crcreg \leftarrow \text{CRC}(a_0 \cdots a_{l-N-1}) \oplus \text{FINALXOR}$
$a_{l-N} \cdots a_{l-1} \leftarrow \left(tcrcreg \odot \left(x^N\right)^{-1} \oplus crcreg\right) \mod p_{\text{CRC}}(x)$
Output: a (bits at the end are adjusted)

Note: This all can be done again a bit more simply if you can afford the space for another table, the "reverse CRC table". The way to build it and use it for the purpose of what we did here with the inverse polynomial is described in the following section, especially in its "improved" version.

5 Getting a chosen CRC by altering a chosen position

5.1 Theory behind it

We know from the last sections how the last N bits of our data can be modified to have any value we want in the CRC register after the CRC computation. Now, suppose we want to modify the N bits somewhere else, let's say at position $k \leqslant l - N$ (where $k = l - N$ would be the position for altering the last N bits of data):

$$\text{crc}(a_0 \cdots a_{k-1} \underbrace{\tilde{a}_0 \cdots \tilde{a}_{N-1}}_{\text{modified}} a_{k+N} \cdots a_{l-1}) = r_0 \cdots r_{N-1}$$

If we could determine which value r' in the CRC register is needed right before processing $a_{k+N} \cdots a_{l-1}$ to have a final CRC value of r:

$$\text{crc}(r', a_{k+N} \cdots a_{l-1}) = r$$

...then this is solved by using the method from the previous section, applied to $a_0 \cdots a_{k-1}$ and r', because we presented a method to calculate $\tilde{a}$ for given $a_0 \cdots a_{k-1}$ and r' so that:

$$\text{crc}(a_0 \cdots a_{k-1} \tilde{a}_0 \cdots \tilde{a}_{N-1}) = r'$$

What we need is an algorithm to calculate the value of r' for given values of r and b which solves the equation $\text{crc}(r', b) = r$. In section 4 we calculated b for

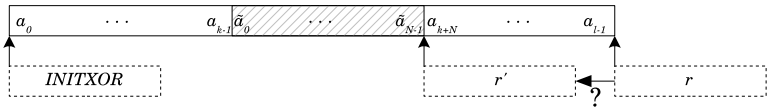

Figure 5: Backwards calculating the CRC

given r' and r. The CRC algorithm itself calculates r for given r' and b, so we just have to apply the CRC algorithm "backwards" (see fig. 5).

To achieve this, remember how the table-driven approach to calculate the CRC worked: (The following is illustrated in figure 6.) In each step, the CRC register (let's call it $c_0 \cdots c_{M-1}$ here) was first left-shifted by M bits. The bits just shifted out were xored with the M bits of the data word, and the result was used as an index to the CRC table. The N bits found in the CRC table were finally xored to the CRC register. Actually, all the $M + N$ bits (index plus mask from table entry) were xored (call this bit-mask $x_0 \cdots x_{M+N-1}$ here, with $x_0 \cdots x_{M-1}$ being the index and $x_M \cdots x_{N+M-1}$ the table entry) to the bits just shifted out and the CRC register itself, but those leading M bits were discarded afterwards, because they were equal to those of the data stream (see 2 for the details).

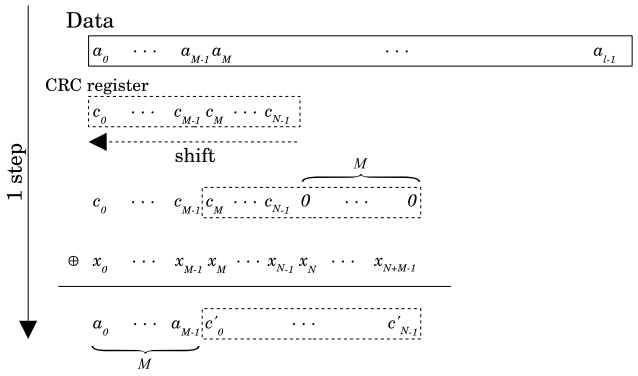

Figure 6: One step of the table-driven CRC calculation

These steps are to be reversed. First, we have to reconstruct the $M+N$ bits that were xored ($x_0 \cdots x_{M+N-1}$). Note that the CRC algorithm shifted M bits left just before the xor, so M 0-bits were shifted in to the right, meaning that the M rightmost bits after the xor ($c'_{N-M} \cdots c'_{N-1}$) are equal to the pattern which was xored to them ($x_N \cdots x_{N+M-1}$). But what pattern was it as a whole? The pattern was created by xoring the CRCPOLY at certain positions. We can recreate it step-by-step by looking at the rightmost M bits of the CRC register, and building a $M + N$ pattern by xoring the CRCPOLY (plus its originally omitted leading 1) at the corresponding positions, beginning with the rightmost.

The resulting pattern is the one that was effectively xored to the shifted-out bits and the CRC register. To speed this process up, all possible patterns can

be stored in a table which we call the "reverse CRC table" and which is indexed over the M rightmost bits $x_N \cdots x_{N+M-1}$ of the xor pattern $x_0 \cdots x_{M+N-1}$ (instead of the leftmost bits as in the CRC table). Therefore, the resulting table is of the same size as the CRC table itself.

Having the correct pattern of $M + N$ bits, we can xor it, but we keep only the leftmost N bits of the result in the new, "restored" CRC register (because the M rightmost bits are by definition 0 after that). This also reversed the left-shifting. Finally, the M data bits have to be xored to the leftmost M bits of the CRC register, so that the CRC register is restored as it was right before a CRC calculating step. Just "read" figure 6 bottom-up for a better understanding of this process.

5.2 Pseudo-code

Building the "reverse CRC table" and using it to alter the chosen data bits is described in the following pseudo-code. Its structure is somewhat similar to those of the table-driven approach to calculate the CRC itself. (Note that we use CRCPOLY_{-1} to denote the omitted leading coefficient of $p_{\text{CRC}}(x)$ which is always 1. Note further that we use a function $\textsc{AdjustData}(r', r)$ which represents the part from section 4 which calculated the needed data bits b so that $\text{crc}(r', b) = r$.)

Algorithm 5 Calculating the reverse CRC table

Input: (nothing)
  for $index = 0$ to $2^M - 1$ **do**
    $crcreg \leftarrow 0$ {Note that $crcreg$ is N bits width!}
    $crcreg_{N-M} \cdots crcreg_{N-1} \leftarrow index_0 \cdots index_{M-1}$
    for $k = 1$ to M **do**
      $\textsc{RightShift}(crcreg)$
      if $bit\_just\_shifted\_out = 1$ **then**
        $crcreg \leftarrow crcreg \oplus \text{CRCPOLY}_{-1} \cdots \text{CRCPOLY}_{N-2}$
      end if
    end for
    $revcrctable[index] \leftarrow crcreg$
  end for
Output: $revcrctable$

Algorithm 6 $\textsc{bwCrc}()$ – Table-driven "backwards" calculation of the CRC

Input: a (containing the data bits), $tcrcreg$ (wanted CRC)
  $tcrcreg \leftarrow tcrcreg \oplus \text{FINALXOR}$
  for $i = (l/M) - 1$ downto 0 **do**
    $\textsc{RightShift}(trcreg, M)$
    $index \leftarrow bits\_just\_shifted\_out$
    $tcrcreg \leftarrow tcrcreg \oplus revcrctable[index]$
    $tcrcreg_0 \cdots tcrcreg_{M-1} \leftarrow tcrcreg_0 \cdots tcrcreg_{M-1} \oplus a_i \cdots a_{i+M-1}$
  end for
Output: $tcrcreg$

Algorithm 7 Data adjustmend at a chosen position

Input: a (containing the data bits), $tcrcreg$ (wanted CRC), k (chosen position)

$crcreg \leftarrow \text{CRC}(a_0 \cdots a_{k-1}) \oplus \text{FINALXOR}$
$tcrcreg \leftarrow \text{BWCRC}(a_{k+N} \cdots a_{l-1}, tcrcreg)$
$a_k \cdots a_{k+N-1} \leftarrow \text{ADJUSTDATA}(crcreg, tcrcreg)$

Output: a (bits at position k are adjusted)

Note that this last algorithm is replaced by a more elegant version (without the need of ADJUSTDATA()) in the following subsection.

5.3 Improving this approach

In short we did the following (see also fig. 7):

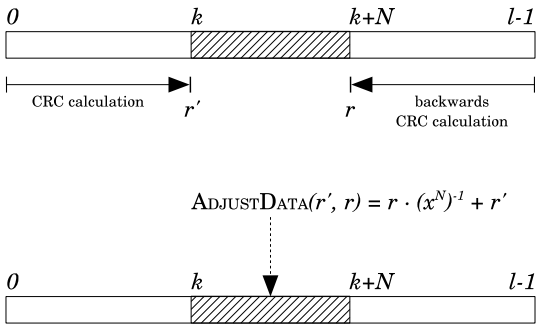

Figure 7: Old approach to adjust the data at a chosen position

1. Calculate the crc from the beginning up to position k, call it r'. (This is done by the standard CRC algorithm.)
2. Calculate the crc backwards from the end up to position $k + N$, call it r. (This was done above in section 5.1.)
3. Calculate new bits $\tilde{a}$ with $\text{crc}(r', \tilde{a}) = r$ and inject them into your data at position k. (See section 4 for calculating that.)

As we will see, the last step can also be done within step 2, resulting in a very simple algorithm. First remember section 5.1, which can calculate r' for given values of r and b, so that:

$$\text{crc}(r', b) = r$$

Second, remember what we did in section 4, we looked for $\tilde{a}$ while r and r' where given, so that:

$$\text{crc}(r', \tilde{a}) = r \tag{5}$$

Looks quite similar, but the variables we look for are different for both cases (r' in the first, $\tilde{a}$ in the latter). Let's see if we can do something about that.

Remember further (from section 3.2), that for all x with a width of N we have:

$$\forall x : \mathrm{crc}(x, x) = 0 \tag{6}$$

And finally remember the property of $\mathrm{crc}(.,.)$ which we discovered in 3.2 while analysing the "magic sequence" m:

$$\mathrm{crc}(r_1, a_1) \oplus \mathrm{crc}(r_2, a_2) = \mathrm{crc}(r_1 \oplus r_2, a_1 \oplus a_2) \tag{7}$$

What we do now is to use $r' \oplus \tilde{a}$ for x in equation (6) and add this instance of (6) to (5) while using the homomorphical property of crc described by (7):

$$\begin{aligned} r &= \mathrm{crc}(r', \tilde{a}) \oplus \mathrm{crc}(r' \oplus \tilde{a}, r' \oplus \tilde{a}) \\ &\overset{(7)}{=} \mathrm{crc}(r' \oplus r' \oplus \tilde{a}, \tilde{a} \oplus r' \oplus \tilde{a}) \\ &= \mathrm{crc}(\tilde{a}, r') \end{aligned}$$

Wow! We see that $\mathrm{crc}(r', \tilde{a}) = \mathrm{crc}(\tilde{a}, r')$. So how does this help? Well, look at step 3, were r and r' are given and $\tilde{a}$ is computed. As we know now, we can equivalently write:

$$\mathrm{crc}(r', \tilde{a}) = r \quad \Longleftrightarrow \quad \mathrm{crc}(\tilde{a}, r') = r$$

So if we are looking for $\tilde{a}$ when r' and r are given, we can interpret the unknown $\tilde{a}$ as the initial CRC register, the given r' as the data to be computed, and (as before) the also given r as the final CRC register. Because we are looking for $\tilde{a}$, this can easily be done by step 2, our "backwards" CRC algorithm! This means that we don't need step 3 anymore (so the method from section 4 is not really needed, if you have the pre-computed "reverse CRC table" available) and can instead do the following (see also fig. 8):

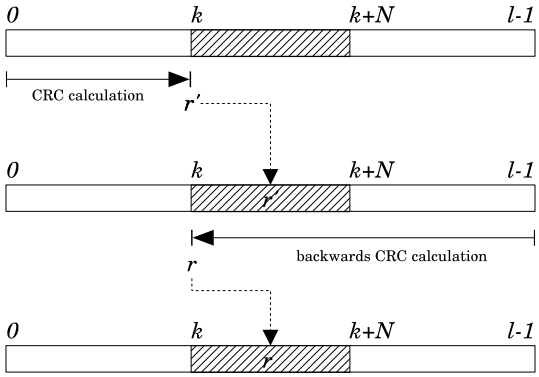

Figure 8: Improved approach to adjust the data at a chosen position

1. Calculate the crc from the beginning up to position k using the standard CRC algorithm, and call it r'. Then overwrite the N data bits at position k with this r'.

2. Calculate backwards the crc from the end up to position k (!) using our "backwards CRC" algorithm from section 5.1, and call it r. Overwrite the N data bits at position k again, but this time use r for this.

5.4 Pseudo-code

Finally, this are the two lines of pseudo-code, summarizing all we did up to this section. Note that the goal of section 4 can be considered as a special case of this, where $k = l - N$. This renders AdjustData() obsolete.

Algorithm 8 Data adjustmend at a chosen position, improved version

Input: a (containing the data bits), $tcrcreg$ (wanted CRC), k (chosen position)

$a_k \cdots a_{k+N-1} \leftarrow \text{Crc}(a_0 \cdots a_{k-1}) \oplus \text{FINALXOR}$
$a_k \cdots a_{k+N-1} \leftarrow \text{bwCrc}(a_k \cdots a_{l-1}, tcrcreg)$

Output: a (bits at position k are adjusted)

6 Conclusion

The presented methods offer a very easy and efficient way to modify your data so that it will compute to a CRC you want or at least know in advance. This is not a very difficult task, as CRC is not a cryptographical hash algorithm – it was never meant to be one. So you should *never* consider the CRC as some kind of message authentication code (like some of the copy-protection guys do) – it can easily be forged.

The fact that the CRC can be forged really easily makes one think of other applications of our algorithms. It could be used as some sort of "covert channel" for the undetected transmission or storage of data. You could hide data in the CRC of other data which itself could look unsuspicious. An interesting research topic could be how to extend our methods so that the modified bits don't have to be in one block but could be spread over the whole data stream. Our intuition suggests that this could be a possible (but non-trivial) task. Feel free to do some research in this area, it could be fun!

A Appendix

This appendix contains the source code of the implementation of our algorithms for the CRC32 in the C programming language. Refer to the according sections to get a detailed description and reasoning. This code will compile with every modern C compiler (and will work correctly as long as you adjust the definition of `uint32` so that it's an unsigned integer with 32 bits). All functions are relatively small in size, as one of our goals was to provide fast and simple solutions.

A.1 Definitions

We use some pre-defined values within our C code. We collect them altogether here at the beginning, so that every piece of code will directly compile if you prepend these definitions. Don't forget to adjust `uint32` which is meant to be an unsigned integer with 32 bits. This could be expressed differently on your system, so please consult the documentation of your C compiler if necessary.

```
#define CRCPOLY 0xEDB88320
#define CRCINV 0x5B358FD3 // inverse poly of (x^N) mod CRCPOLY
#define INITXOR 0xFFFFFFFF
#define FINALXOR 0xFFFFFFFF
typedef unsigned int uint32;
```

Listing 1: Definitions

A.2 Bit-oriented implementation of CRC32

Our first implementation is the bit-oriented approach of CRC32. This is of less interest for real-world applications, at it is outperformed by the table-driven implementation by at least a factor of 8, but could be interesting for a deeper understanding of how the CRC works. It may also come handy if there's no pre-built CRC table (e.g. no room for it), but these are probably rare cases ...

```
/**
 * Computes the CRC32 of the buffer of the given length using the
 * (slow) bit-oriented approach
 */
int crc32_bitoriented(unsigned char *buffer, int length) {
    int i, j;
    uint32 crcreg = INITXOR;

    for (j = 0; j < length; ++j) {
        unsigned char b = buffer[j];
        for (i = 0; i < 8; ++i) {
            if ((crcreg ^ b) & 1) {
                crcreg = (crcreg >> 1) ^ CRCPOLY;
            } else {
                crcreg >>= 1;
            }
            b >>= 1;
        }
    }

    return crcreg ^ FINALXOR;
}
```

Listing 2: Bit-oriented implementation of CRC32

A.3 Table-driven implementation of CRC32

This implementation of CRC32 is similar to the one used everywhere. First, the CRC32 table is built and then it can be used to call the actual CRC32 calculating function as often as needed.

```
/**
 * Creates the CRC table with 256 32-bit entries. CAUTION: Assumes that
 * enough space for the resulting table has already been allocated.
 */
void make_crc_table(uint32 *table) {
    uint32 c;
    int n, k;

    for (n = 0; n < 256; n++) {
        c = n;
        for (k = 0; k < 8; k++) {
            if ((c & 1) != 0) {
                c = CRCPOLY ^ (c >> 1);
            } else {
                c = c >> 1;
            }
```

```
        }
        table[n] = c;
    }
}
```

Listing 3: (Pre-)Building the CRC32 table

```
/**
 * Computes the CRC32 of the buffer of the given length
 * using the supplied crc_table
 */
int crc32_tabledriven(unsigned char *buffer,
                      int length,
                      uint32 *crc_table)
{
    int i;
    uint32 crcreg = INITXOR;

    for (i = 0; i < length; ++i) {
        crcreg = (crcreg >> 8) ^ crc_table[((crcreg ^ buffer[i]) & 0xFF)];
    }

    return crcreg ^ FINALXOR;
}
```

Listing 4: Table-driven implementation of CRC32

A.4 Data adjustment at the end for a known CRC

Now for the first data adjustment: As the title suggests, the buffer will compute to a known CRC afterwards (see section 3 for details), which will be the "magic sequence" `0x2144DF1C` for CRC32. This is done by just appending the CRC32 of all bytes except the last four.

```
/**
 * Changes the last 4 bytes of the given buffer so that it afterwards will
 * compute to the "magic sequence" (usually 0x2144DF1C for CRC32)
 */
void fix_crc_magic(unsigned char *buffer, int length, uint32 *crc_table)
{
    int i;

    // calculate CRC32 except for the last 4 bytes
    uint32 crcreg = crc32_tabledriven(buffer, length-4, crc_table);

    // inject crcreg as content - nothing easier than that!
    for (i = 0; i < 4; ++i)
        buffer[length - 4 + i] = (crcreg >> i*8) & 0xFF;
}
```

Listing 5: Implementation of data adjustment at the end for a known CRC

A.5 Data adjustment at the end for a chosen CRC

Second, the implementation of adjusting the last four bytes so that the buffer calculates to the chosen CRC afterwards. This is achieved by calculating the CRC32 of all bytes except the last four (which is done by the table-driven implementation of CRC32 for which you have to supply a pointer to the pre-calculated CRC table). After that, some multiplication and addition within the ring of polynomial congruence classes modulo $p_{\mathrm{CRC}}(x)$ is done, which sounds more complicated than it actually is. See section 4 for details.

```
/**
 * Changes the last 4 bytes of the given buffer so that it afterwards will
 * compute to the given tcrcreg using the given crc_table
 *
 * This function uses the method of the multiplication with (x^N)^-1.
 */
void fix_crc_end(unsigned char *buffer,
                 int length,
                 uint32 tcrcreg,
                 uint32 *crc_table)
{
    int i;
    tcrcreg ^= FINALXOR;

    // calculate crc except for the last 4 bytes; this is essentially crc32()
    uint32 crcreg = INITXOR;
    for (i = 0; i < length - 4; ++i) {
        crcreg = (crcreg >> 8) ^ crc_table[((crcreg ^ buffer[i]) & 0xFF)];
    }

    // calculate new content bits
    // new_content = tcrcreg * CRCINV mod CRCPOLY
    uint32 new_content = 0;
    for (i = 0; i < 32; ++i) {
        // reduce modulo CRCPOLY
        if (new_content & 1) {
            new_content = (new_content >> 1) ^ CRCPOLY;
        } else {
            new_content >>=  1;
        }
        // add CRCINV if corresponding bit of operand is set
        if (tcrcreg & 1) {
            new_content ^= CRCINV;
        }
        tcrcreg >>= 1;
    }
    // finally add old crc
    new_content ^= crcreg;

    // inject new content
    for (i = 0; i < 4; ++i)
        buffer[length - 4 + i] = (new_content >> i*8) & 0xFF;
}
```

Listing 6: Implementation of data adjustment at the end for a chosen CRC

A.6 Data adjustment at chosen position for a chosen CRC

Finally, this is the implementation of our method to adjust four bytes at some chosen position within the buffer so that the buffer calculates to the chosen CRC afterwards. All that is needed for this is some forward-calculating of the CRC (using the CRC table) and then some backwards-calculating (using a reverse CRC table). Please consult section 5 for details. We also provide the function to pre-calculate the reverse CRC table, which is also described in section 5. Note that the parameter `fix_pos`, which marks the position of the bytes to be adjusted, can also be negative, which is then counted from the end of your buffer (so that a value of `-4` will result in adjusting the last 4 bytes). Have fun!

```
/**
 * Creates the reverse CRC table with 256 32-bit entries. CAUTION: Assumes
 * that enough space for the resulting table has already been allocated.
 */
void make_crc_revtable(uint32 *table) {
    uint32 c;
    int n, k;
```

```
    for (n = 0; n < 256; n++) {
        c = n << 3*8;
        for (k = 0; k < 8; k++) {
            if ((c & 0x80000000) != 0) {
                c = ((c ^ CRCPOLY) << 1) | 1;
            } else {
                c <<= 1;
            }
        }
        table[n] = c;
    }
}
```

Listing 7: (Pre-)Building the reverse CRC32 table

```
/**
 * Changes the 4 bytes of the given buffer at position fix_pos so that
 * it afterwards will compute to the given tcrcreg using the given crc_table.
 * A reversed crc table (crc_revtable) must be provided.
 */
void fix_crc_pos(unsigned char *buffer,
                 int length,
                 uint32 tcrcreg,
                 int fix_pos,
                 uint32 *crc_table,
                 uint32 *crc_revtable)
{
    int i;
    // make sure fix_pos is within 0..(length-1)
    fix_pos = ((fix_pos%length)+length)%length;

    // calculate crc register at position fix_pos; this is essentially crc32()
    uint32 crcreg = INITXOR;
    for (i = 0; i < fix_pos; ++i) {
        crcreg = (crcreg >> 8) ^ crc_table[((crcreg ^ buffer[i]) & 0xFF)];
    }

    // inject crcreg as content
    for (i = 0; i < 4; ++i)
        buffer[fix_pos + i] = (crcreg >> i*8) & 0xFF;

    // calculate crc backwards to fix_pos, beginning at the end
    tcrcreg ^= FINALXOR;
    for (i = length - 1; i >= fix_pos; --i) {
        tcrcreg = (tcrcreg << 8) ^ crc_revtable[tcrcreg >> 3*8] ^ buffer[i];
    }

    // inject new content
    for (i = 0; i < 4; ++i)
        buffer[fix_pos + i] = (tcrcreg >> i*8) & 0xFF;

}
```

Listing 8: Implementation of data adjustment at chosen position for a chosen CRC

References

[ana99] anarchriz. CRC and how to reverse it, April 1999. Available from http://www.woodmann.com/fravia/crctut1.htm.

[PSMR06] Henryk Plötz, Martin Stigge, Wolf Müller, and Jens-Peter Redlich. Self-Replication in J2ME MIDlets. May 2006. Available from http://sar.informatik.hu-berlin.de/research/publications/#SAR-PR-2006-04.

[Tan81] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

[Wes03] Bas Westerbaan. The breaking of cyber patrol 4, March 2003. Available from http://www.cs.cmu.edu/~dst/CP4break/cp4break.html#sec4.

[Wes05] Bas Westerbaan. Reversing crc, July 2005. Available from http://blog.w-nz.com/archives/2005/07/15/reversing-crc/.

[Wil96] Ross N. Williams. A painless guide to crc error detection, September 1996. Available from http://www.repairfaq.org/filipg/LINK/F_crc_v3.html.

Reports published by Humboldt University Berlin, Computer Science Department, Systems Architecture Group.

---

1. SAR-PR-2005-01: Linux-Hardwaretreiber für die HHI CineCard-Familie. Robert Sperling. 37 Seiten.
2. SAR-PR-2005-02, NLE-PR-2005-59: State-of-the-Art in Self-Organizing Platforms and Corresponding Security Considerations. Jens-Peter Redlich, Wolf Müller. 10 pages.
3. SAR-PR-2005-03: Hacking the Netgear wgt634u. Jens-Peter Redlich, Anatolij Zubow, Wolf Müller, Mathias Jeschke, Jens Müller. 16 pages.
4. SAR-PR-2005-04: Sicherheit in selbstorganisierenden drahtlosen Netzen. Ein Überblick über typische Fragestellungen und Lösungsansätze. Torsten Dänicke. 48 Seiten.
5. SAR-PR-2005-05: Multi Channel Opportunistic Routing in Multi-Hop Wireless Networks using a Single Transceiver. Jens-Peter Redlich, Anatolij Zubow, Jens Müller. 13 pages.
6. SAR-PR-2005-06, NLE-PR-2005-81: Access Control for off-line Beamer – An Example for Secure PAN and FMC. Jens-Peter Redlich, Wolf Müller. 18 pages.
7. SAR-PR-2005-07: Software Distribution Platform for Ad-Hoc Wireless Mesh Networks. Jens-Peter Redlich, Bernhard Wiedemann. 10 pages.
8. SAR-PR-2005-08, NLE-PR-2005-106: Access Control for off-line Beamer Demo Description. Jens Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge. 28 pages.
9. SAR-PR-2006-01: Development of a Software Distribution Platform for the Berlin Roof Net (Diplomarbeit / Masters Thesis). Bernhard Wiedemann. 73 pages.
10. SAR-PR-2006-02: Multi-Channel Link-level Measurements in 802.11 Mesh Networks. Mathias Kurth, Anatolij Zubow, Jens Peter Redlich. IWCMC 2006 - International Conference on Wireless Ad Hoc and Sensor Networks, Vancouver, Canada, July 3-6, 2006.
11. SAR-PR-2006-03, NLE-PR-2006-22: Architecture Proposal for Anonymous Reputation Management for File Sharing (ARM4FS). Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge, Torsten Dänicke. 20 pages.
12. SAR-PR-2006-04: Self-Replication in J2me Midlets. Henryk Plötz, Martin Stigge, Wolf Müller, Jens-Peter Redlich. 13 pages.
13. SAR-PR-2006-05: Reversing CRC – Theory and Practice. Martin Stigge, Henryk Plötz, Wolf Müller, Jens-Peter Redlich. 24 pages.