B Theory and Practice of CRCs

This section is devoted to the general theory and practice of CRCs. The problem that we try to solve with the use of CRCs is the following: Given a long sequence of bit, compute from it a short sequence of bit such that two short sequences are different if they were computed from two different long sequences. The solution of this problem can be used to check a long sequence of bit for transmission errors simply by comparing two short sequences, one computed before the transmission and one after. But, of course, this problem has no solution since there are many more long sequences than short sequences, and hence, not all long sequences can be mapped to different short sequences. So we have to modify the problem in order to solve it. We can not really expect to be able to detect any kind of transmission error with only the very limited information provided by a short sequence of check bit. But we can reasonably expect to be able to find at least the most common errors, and the most common errors are small errors. Therefore we have the new problem: Given a long sequence of bit, compute from it a short sequence of bit such that two short sequences are different if they are computed from two long sequences that differ only by a few bit.

This is a rather difficult problem because of its combinatory complexity. Even if we consider only two single bit errors, there are already many, many different possibilities how these two errors can occur, for example in the 38 protected byte of a version 1, layer III frame. Hence, without some mathematical theory, there is no way of deriving a solution. The theory, we will use, is the theory of polynoms over the Field of the two numbers 0 and 1.

B.1 Fields and Rings

When we do computations with numbers like 1, -3, $\frac{1}{7}$, or 3.1415, we use certain rules of computation, like $a+b=b+a$, $a\cdot 1=a$, or $a+(-a)=0$. The complete set of rules does not matter for our informal discussion here, but a good book on algebra (e.g. the timeless classic [32] or [13]) is recommended for those with a deeper interest in the subject. Sets of numbers that fulfill these rules are called Fields. For example, the set of real numbers is such a Field. It might come as a surprise that there are small finite sets of numbers that satisfy all the normal rules of computation and therefore are Fields.

To further investigate the topic, think, for example, about the numbers from 1 to 12, as we use them for the 12 hours on a clock. We can do all the normal computations

on these because we restart with 1 each time a number gets bigger than 12. For example, 10 plus 4 is 2 o'clock. Multiples of 12 hours do not matter in regard to the hour shown on a clock and are neglected. We only care for the remainder after discarding any multiples of 12, and two numbers are considered equal if they differ only by a multiple of 12. The number 12 itself is like 0. Mathematically we say: we compute "modulo 12".

But, do we have negative numbers? Look at the equation $10+4=2$ that we stated above. In this equation, the number 4 behaves like -8. Instead of going 8 hours backward, we can always go 4 hours forward and the clock will show he same time. It is easy to see that for each number on the clock there is also the negative number present.

How about fractions? Is there a number like $\frac{1}{5}$ with the property $5 \cdot \frac{1}{5} = 1$? In this case the answer is yes. We have $5 \cdot 5 = 25 = 12 + 12 + 1 = 1$, and the number 5 takes the role of $\frac{1}{5}$. We should not expect to find a number for $\frac{1}{12}$ since $12 = 0$ and there is no such thing as $\frac{1}{0}$. There is, however, also no number for $\frac{1}{2}$. Whatever number a we choose, $2 * a$ will always be an even number and it can never be 1 or a multiple of 12 plus 1 which is always an odd number. We conclude that the numbers modulo 12 are not a Field; they constitute what mathematicians call a Ring. A Ring is an algebraic structure, where we can do addition, subtraction, and multiplication as usual, but division is limited. Division might leave a remainder. The whole numbers, for example, constitute a Ring. A deeper investigation reveals, that the whole numbers modulo n constitute a Field, if and only if n is a prime number. The numbers modulo 2 constitute the simplest and smallest Field. It is called G_2 and consists only of the two numbers 0 and 1, which we can also interpret as the boolean values "false" and "true". Multiplication and addition can then be written as $\wedge$ (and) and $\oplus$ (exclusive or), because the rules of computation are the same for these operations.

The Field G_2 offers us a new mathematical model for sequences of bit: bit vectors. So far we have seen vectors only as vectors of real numbers, but now, we know that the boolean values 0 and 1 obey the same rules of computation as the real numbers, and we can easily reuse the theory of vectors for vectors of booleans, or in general vectors over an arbitrary Field. Unfortunately linear algebra is not sufficient for the theory of CRCs, and we have to go one step further. We have to consider polynoms.

B.2 Polynom Rings

A polynom p over the variable x is an expressions of the form

$$p = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 x^0.$$

The numbers $a_n, a_{n-1}, \ldots, a_1, a_0$ are taken from a fixed but otherwise arbitrary Field and are called "coefficients". If $a_n \neq 0$ we call n the degree of the polynom. Most people know polynoms like $x^2 - 2x + 1$ from school. Here, we consider polynoms with coefficients from the Field G_2.

Since polynoms are added component wise and multiplication by numbers from the Field is again component wise, we know that polynoms form a vector space. The

polynom above can be equally well written as the vector $(a_n, a_{n-1}, \ldots + a_1, a_0)$. There is, however, much more we can do with a polynom: We can multiply polynoms, and we can evaluate polynoms for any value of x taken from the Field. We can even divide one polynom by another, but usually this leaves a remainder, and hence polynoms are not a Field, but a Ring.

The Ring of polynoms over the Field G_2 is the right algebraic model to use if we want to study CRCs. A sequence of bit like 101101 is modeled by the polynom $1x^5 + 0x^4 + 1x^3 + 1x^2 + 0x^1 + 1x^0$ which we can write shorter as $x^5 + x^3 + x^2 + 1$.

The CRC of a bit sequence is defined using a so called generator polynom. In the case of MPEG streams this generator polynom is

$$g = x^{16} + x^{15} + x^2 + 1.$$

The definition of the CRC for a bit sequence is now very simple: We take the polynom p that belongs to the given bit sequence, divide it by g, and obtain a remainder. The bit sequence that corresponds to this remainder is the CRC.

This definition of the CRC is a clever move: it yields a code that can be computed efficiently in hardware and in software, as we will see below, and lends itself to a thorough investigation of the properties of such a code.

B.3 Properties

Without too much mathematics, we can derive some of the properties of CRCs with simple means. More elaborate techniques can be found in the classic CRC paper [26] or in a books like [31] and [20].

Assume a polynom p and a polynom $\tilde{p} \neq p$, which differ by some bit caused by a transmission error. We investigate the question: Under what circumstances will that error pass by undetected, that is, under what circumstances will both polynoms have the same remainder when divided by the generator polynom g? We can write $p = qg + r$ and $\tilde{p} = \tilde{q}g + \tilde{r}$, for suitable factors q and $\tilde{q}$ and remainders r and $\tilde{r}$. If both polynoms yield the same CRC, we have $r = \tilde{r}$ and we can conclude $p - \tilde{p} = (qg + r) - (\tilde{q}g + \tilde{r}) = (q - \tilde{q})g + (r - \tilde{r}) = (q - \tilde{q})g$.

The last equation states the first important property of CRCs:

Property 1: Two bit sequences yield the same CRC, if and only if the difference of the two sequences is a multiple of the generator polynom g.

Now we apply this to the case of a single bit error. The situation is as follows:

$$\begin{array}{rr} & \downarrow \\ p & \ldots 011010101 \ldots \\ -\tilde{p} & -\ \ldots 011000101 \ldots \\ \hline p - \tilde{p} & \ldots 000010000 \ldots \end{array}$$

Subtracting $\tilde{p}$ from p will zero out all the coefficients, except at the position (indicated by the arrow) where the two bit sequences differ. So we have $p - \tilde{p} = x^k$ where k is the position of the bit error, and we have reduced our investigation

to the simple mathematical question: is x^k a multiple of g. In our case, where $g = x^{16} + x^{15} + x^2 + 1$, the answer is: No. The only factors of x^k are $1, x, x^2, \ldots, x^k$ and nothing else. We conclude:

Property 2: Any generator polynom g with more than one term can detect any single bit error.

How about two error bit? Can one error bit cancel out the effect of the other error bit on the CRC? If p and $\tilde{p}$ differ by two bit, we have $p - \tilde{p} = x^k + x^\ell$, where k and ℓ are the positions of the error bit. Then the equation $x^k + x^\ell = (x^{k-\ell} + 1) \cdot x^\ell$ reveals that the two errors will go by undetected, if g either divides $x^{k-\ell} + 1$ or x^ℓ. If we exclude, as above, the second possibility, we can conclude that two single bit errors will remain undetected if the distance $d = k - \ell$ between the two error bit is such that $x^d + 1$ is a multiple of g. One idea to determine the smallest such d would be to try successively $d = 1, 2, 3, \ldots$, compute the remainder, and see whether it is 0. Unfortunately, this would take a long long time. Some more mathematics—learning it will take some time too, but is a lot more fun—reveals that our g has two factors, $g = (x + 1)(x^{15} + x^{14} + x + 1)$, of which the second factor $x^{15} + x^{14} + x + 1$ is a so called primitive polynom. Primitive polynoms have a nice property: for them, the smallest d such that $x^d + 1$ is a multiple of the primitive polynom is $d = 2^n - 1$, where n is the degree of the primitive polynom. And this is as good as it can get, because for all polynoms of degree n, the minimal d is a factor of $2^n - 1$ and hence can not be larger.

Property 3: If the generator polynom g is a primitive polynom of degree n, it can detect two single bit errors as long as the distance between the two errors is less than $2^n - 1$.

In our case, the smallest d such that $x^d + 1$ is a multiple of $x^{15} + x^{14} + x + 1$ is $d = 2^{15} - 1 = 32767$. As long as the distance of the two single bit errors is less than 32767 bit ($\approx$ 4 kbyte), our CRC will detect them. Since an MPEG audio frame easily fits into 4 kbyte, the CRC guards them against two single bit errors.

How about three error bit? A general method, used often with polynoms, can help us to gain clarity. If we have $x^k + x^\ell + x^j = hg$, we can evaluate the polynoms on both sides of the equation for the same value x and get the same result. If we evaluate both sides for the value $x = 1$, we have: $1^k + 1^\ell + 1^j = 1 + 1 + 1 = 1$ and $h(1) \cdot g(1) = h(1) \cdot (1^{16} + 1^{15} + 1^2 + 1) = h(1) \cdot (1 + 1 + 1 + 1) = h(1) \cdot 0 = 0$. And we conclude, that the two sides must be different for any polynom h. So all three bit errors are detectable and in general all errors with an odd number of error bit are detectable.

Property 4: If the generator polynom g has an even number of terms, it can detect any odd number of bit errors.

How about four or more bit? Don't expect too much. Since the error pattern 1 1000 0000 0000 0101 corresponds exactly to the generator polynom $x^{16} + x^{15} + x^2 + x$, it is obviously a multiple of g and we have a simple example of a four bit error that can not be detected. On the other hand, any error pattern that is shorter than this is represented by a polynom of degree 15 or less and therefore can not be a multiple of

g, since multiplying with a polynom (except with the polynom $x^0 = 1$) will increase the degree. So our CRC will detect all error pattern that are confined within 16 bit (or 2 consecutive byte) and the only undetectable error pattern of length 17 is g itself.

Property 5: If the generator polynom g has degree n, it can detect any burst of error bit that is confined to a single stretch of no more then n bit. The only burst error pattern of length $n+1$ that can not be detected is the generator polynom itself.

If our error pattern is spread out more than 17 bit, some pattern, but not very many, are undetectable.

Property 6: If the generator polynom g has degree n, it will fail to detect a burst of error bit longer than $n+1$ bit with a likelihood of only 2^{-n}.

For our generator polynom of degree 16, the fraction of undetectable error patterns is exactly $1/2^{16}$ or 0.0015 %.

Now that we have seen, that our CRC provides the best error protection that we can buy with only 16 bit, we study how to compute it efficiently.

B.4 Bit Wise Computation

To compute the CRC, we have to divide the polynom p representing the given bit sequence by g, the generator polynom, and take the remainder. As we do when we manually perform a division, we use an incremental algorithm. An example will illustrate this: Assume we divide, using decimal numbers, 2301 by 19. How do we go about it?

First, we look at the head of the number: 2 and decide how often $19\cdots$ will "fit in" $20\cdots$. Once! So we subtract 19 from 20 and obtain the remainder 01 Actually, we were subtracting 1900, a multiple of 19, from 2000 and get a remainder of 100. But we don't care about the thousands, hundreds, or $\cdots$, all we care about is that we are now done with the thousands and continue with the hundreds. That is, we can now shift the result 01 one place to the left and get 1, computing now in units of one hundred, not one thousand. We consider the next digit 3 (hundred). Before we use it, we add the remainder from the previous computation: $3+1 = 4$ (hundred). Then the algorithm repeats: We look at the 4 and decide how often $19\cdots$ will "fit in" $40\cdots$. Two times! We subtract 38 and obtain the remainder 2. We add this remainder with the next digit $2 + 0 = 2$ (actually $20 + 00 = 20$) and continue. We look at 2 and decide that we should again subtract 19 from 20, and get the remainder 1. We add the remainder 1 and the next digit 1, to give a total remainder of 2. We stop here, since shifting is no longer possible because we have reached the end of the dividend. 2 is the final result.

With polynoms, we follow the same algorithm. Lets divide

$$1x^5 + 0x^4 + 0x^3 + 1x^2 + 0x + 0 \quad \text{by} \quad 1x^2 + 0x + 1.$$

We consider the leading "digit" $1x^5$ and see that $1x^2 + 0x + 1$ will fit in x^3 times. This is simple, we just need to compare the two leading exponents. We subtract $(1x^2 + 0x + 1)x^3$ from $1x^5$ and get $0x^5 + 0x^4 + 1x^3$. Note that the highest exponent

has disappeared. We shift, that is discard the $0x^5$, add $0x^4$, the next "digit" from the dividend, to $0x^4 + 1x^3$ and repeat. We look at $0x^4 + 1x^3$ and see that no subtraction is necessary. We shift again, add $0x^3$ (the next "digit"), and repeat. We have $1x^3$ and decide that we must subtract $(1x^2 + 0x + 1)x^1$. The result is: $0x^3 + 0x^2 + 1x$. We shift, add the next "digit", and we have: $1x^2 + 1x$. A last time, we subtract $(1x^2 + 0x + 1)x^0$ and get $0x^2 + 1x + 1$. Shifting is no longer possible and we stop here with the remainder $1x+1$. We were lucky so, that the dividend did not contain further non zero coefficients for x^1 and x^0. Otherwise, we should have added them to the final result. One way to avoid this unnecessary complication, is the multiplication of the original polynom by x^{d-1}, where d is the degree of the generator polynom, before computing the remainder. This will ensure that the last d coefficients are always zero and avoids the extra addition. The loop can then terminate exactly after dealing with the last digit of the original polynom. The properties of the CRC do not change through this modification. Like most applications, the MPEG standard uses the CRC in this modified way.

Computing with polynoms and binary coefficients is even easier than computing with decimal numbers. Whether the dividing polynom "fits in" is decided purely by the leading coefficient. If it is 1, we subtract, otherwise, we don't. We do not really need to compute multiples either. We subtract once or not at all. As an intermediate result, we need to carry along only one polynom, the remainder so far, and its degree is always smaller than the degree of the generator polynom.

The algorithm to compute the (modified) CRC for a bit sequence of length n as the remainder of the corresponding modified polynom is now as follows:

- Shift the variable *crc*, the remainder so far, one bit to the left..
- Align the next input *bit* with bit d of the *crc*, where d is the degree of the generator polynom g, and add.
- Consider bit d, the most significant bit (*msb*) of the sum.
- If it is 1, subtract the generator polynom from the sum otherwise leave it untouched.
- Repeat for all n bit.

⟨ compute the CRC for n bit 438 ⟩ ≡ (438)

```
static unsigned short int
        bitcrc(unsigned short int crc, unsigned short int bit, int n)
{ bit = bit ≪ (16 − n);
  while (n > 0)
  { unsigned short int msb = (bit ⊕ crc) & #8000;
    crc = crc ≪ 1;
    bit = bit ≪ 1;
    if (msb) crc = crc ⊕ #8005;
    n−−;
  }
  return crc;
```

```
}
```

Used in 439 and 440.

The given code uses some optimizations. We do not really need the sum of the shifted *crc* and the next input bit. The addition affects only the *msb* (most significant bit) and if it is not zero anyway, it will be zero after the subtraction. So all we need the *msb* for is to decide about the subtraction. After the subtraction, the highest bit is always zero and is shifted out in the next step. If we change the order of shifting and subtraction, we do not need to compute this highest bit, and restrict the subtraction to the lower bit. Hence it is sufficient to represent the generator polynom without the leading coefficient as `#8005`.

We need this as auxiliar function,

⟨ auxiliary functions 69 ⟩ +≡ (439)
⟨ compute the CRC for n bit 438 ⟩

and for printing.

⟨ printing prerequisites 130 ⟩ +≡ (440)
⟨ compute the CRC for n bit 438 ⟩

B.5 Byte Wise Computation

For performance, we need a faster algorithm that operates on byte not on bit. The algorithm presented here goes back to A. Perez[25]; good tutorials are [34] and [27]. We do not present, however, a general algorithm, but present a specialized solution for the modified CRC with the generator polynom $g = x^{16} + x^{15} + x^2 + x$. Looking for an incremental algorithm that computes the modified CRC not bit for bit, but one byte at a time, we have the following situation:

Assume we have a bit string, or polynom p, and we know already the modified CRC for p, that is, we know r such that for some q, we have $p \cdot x^{16} = qg + r$.

$$\underbrace{01010010\,00\cdots0\,10111011}_{p}\ 00000000\,00000000 = p \cdot x^{16} = qg + r$$

To make the representation as byte and 16-bit words explicit, we use in this section the variables p and q for bit sequences of arbitrary length, the variables r and s for sequences of 16 bit (words) and the variables a, b, c, and d for sequences of 8 bit (byte). The remainder r, for instance, has a maximum degree of 15, and hence fits into a 16-bit word. Because we will need byte wise access to r, we write r as $r = a \cdot x^8 + b$, splitting it into a high-byte a and a low-byte b.

To continue with the problem at hand, we assume that we read in an other byte c and want to know the modified CRC of the bit sequence obtained by appending c to the end of p.

$$\underbrace{01010010\,00\cdots0\,10111011}_{p}\ \underbrace{10110011}_{c}\ 00000000\,00000000 = (px^8 + c)x^{16}$$

Now we use some elementary algebra to obtain

$$\begin{aligned}(px^8 + c)x^{16} &= px^{16}x^8 + cx^{16} \\ &= (qg + r)x^8 + cx^{16} \\ &= qgx^8 + rx^8 + cx^{16} \\ &= qgx^8 + (ax^8 + b)x^8 + cx^{16} \\ &= qgx^8 + ax^{16} + bx^8 + cx^{16} \\ &= qgx^8 + (a + c)x^{16} + bx^8\end{aligned}$$

If we now have a table to look up the modified CRC's for all possible byte values, then we can look up in that table for $a + c$ a 16-bit value s such that

$$(a + c)x^{16} = q'g + s$$

Using this equation, we can continue the above computation and have

$$\begin{aligned}(px^8 + c)x^{16} &= qgx^8 + (a + c)x^{16} + bx^8 \\ &= qgx^8 + q'g + s + bx^8\end{aligned}$$

From the last line, we can read off the new modified CRC as $s + bx^8$. We summarize the algorithm:

To compute the modified CRC of c appended to p, from the modified CRC r of p alone,

- take the high-byte a of r and compute $a + c$;
- look up the modified CRC s for the byte $a + c$;
- take the low-byte b of r and shift it left by 8 bit to obtain bx^8;
- and add bx^8 to s.

This algorithm fits on a single line, taking the next byte c directly from the bit stream as $*byte\_pointer$. In addition the *byte_pointer* gets advanced to the next byte.

⟨ compute the CRC for one byte 441 ⟩ ≡ (441)
  $crc = (crc \ll 8) \oplus crc\_table[(crc \gg 8) \oplus *byte\_pointer{+}{+}]$ Used in 100, 101, and 215.

The table, that contains all the modified CRCs for all byte from 0 to 255 is generated using the *bitcrc* function.

⟨ print element 132 ⟩ +≡ (442)
```
void crc8(int i)
{ printf("0x%04x", bitcrc(0, (unsigned short int) i, 8)); }
```

⟨ print table 135 ⟩ +≡ (443)
```
print_array("static␣unsigned␣short␣const␣crc_table", 256, crc8);
```