CRC (Cyclic Redundancy Check)

1 – Introduction

CRC is one of the methods for transmission error-detection commonly used in network protocols. CRC is also commonly implemented in hardware, so it can be used directly by the lower network layers. The basic principle is: a transmitter sends a check number together with the data in such a way that the receiver can calculate it upon the arrival independently. If the check numbers calculated by both the sender and the receiver are different then a transmission error certainly occurred.

The principle is very simple, but the implementation is not so obvious. A very naive implementation will lead to poor performance. The programming techniques used to improve performance make the program somewhat difficult to read and understand.

Follows a general discussion about CRC. Initially it will be demonstrated how to achieve error detection using decimal arithmetic. Polynomial division is discussed as an alternative to the decimal approach. In the remaining sections there is a brief presentation of bit manipulation in C language, Ethernet CRC generators and performance problems faced by the programmers. Simple examples illustrate the concepts discussed in this text and additional references are given at the end. Those references also contain source examples.

2 – The basic CRC operation using decimal division

Consider a certain message that contains 10 characters. The transmitter and the receiver will have to agree upon a standard *format* and a *generator*.

Both the sender and the receiver need to use the same *format.* This could be simply the location of the original message and the size of the CRC value that needs to be sent as part of the transmitted message. For example, the CRC value is attached to the end of the original message (figure 1).

Figure 1 – A sample: how to combine the original message with its CRC

The *generator* is a number used to divide the [message+CRC] frame. Suppose that the original message is ***M*** and that the transmitter appends a CRC value to form a *frame* ***F***. If this *frame* is divided by the generator ***G***, the remainder of this division should be zero. In other words, the frame ***F*** should be divisible by ***G***. If the receiver finds later that the remainder is different then zero then there was a transmission error. Otherwise nothing can be concluded. However good generators have a very high probability of finding errors if they occur. This is one reason for not using decimal arithmetic in *real* CRC implementations.

In order to find a CRC for a given message one needs to invert the process. The question to be asked is: "*what is the number that should appended to* ***M*** *in order to get a remainder equals to zero when dividing the whole frame by the generator?*" If a number of zeros are appended to the original message ***M*** and the

result is divided by the generator, the difference between the remainder of this division and the generator is the CRC value. For example, lets suppose that a message '10' with a CRC of one digit (generator 3) is to be created:

$$100 \bmod 3 = 1$$

One needs to find a number that appended to 10 yields a number that is divisible by 3:

$3-(100 \bmod 3) = 2$,then 2 is the "CRC value" because

10 appended to 2 = 102 and

$102 \bmod 3 = 0$.

In order to append zeros after the message one needs to shift the characters and add empty bits to fill the equivalent size of the CRC. This is equivalent to multiply ***M*** by powers of 2 (2^n). The exponent *n* indicates how many positions the original message was shifted. However as the characters are represented by hexadecimal numbers, the shift has to be done in hexadecimal (or binary).

Follows a numerical example for a CRC of 16 bits:

M = 'A' (character 'A')

, or ***M*** = 65 and ***G*** = 34943

Appending two bytes to the end of 'A' may be easily done by converting it to hexadecimal.

M = 0x41

After appending 16 bits (or 4 hexadecimal zeros) and converting it back to decimal:

M = 0x410000 = 4259840 (equivalent to 65*2^16)

M mod ***G*** = 31737

Therefore the CRC value is:

CRC = ***G*** – (***M*** mod ***G***) = 34943 – 31737 = 3206 = 0x0C86

The resulting frame ***F*** is:

F = 0x410C86 = 4263046

One can check if ***F*** mod ***G*** yields zero by doing:

4263046 mod 34943 = 0

Now suppose that an error changes a single bit in ***F***, so the receiver receives ***F***' :

F' = 0x420C86 = 4328582

4328582 mod 34943 = 30593 (different then zero!)

The receiver would then know that an error occurred due to the fact that the remainder of ***F***' mod ***G*** is different then zero. Notice also that the value that the receiver may find will be between 0 and ***G*** – 1. This indicates the number of possibilities for a CRC of this size. Every existing message will have some CRC value between these values. Two questions are posed: What is the probability of finding two different messages that yields the same CRC? And how similar are the messages that happen to have the same CRC values? Suppose that an unfortunate coincidence occurs and the error produces a wrong message that fits just the same CRC value. In this case the receiver would not be able to detect any errors. It was found that some generators work better than others because they distribute the CRC values in a way that improves the probability of finding transmission errors.

It is important to stress that such an algorithm, using decimal remainders, would be impractical and are not used in real implementations. The purpose of the example above is to illustrate the process, not to compute a CRC in the standard way. In order to study real CRC algorithms, one needs to learn about polynomial division, in the next section.

3 – Polynomial division

There is a number of reasons why pure decimal division do not work very well in real CRC implementations. Among the reasons is the fact that it is more efficient to use module 2 arithmetic when implementing it in hardware. One needs to understand the polynomial remainder calculation to be able to produce CRC codes that are closer to the ones used in practice.

Given two polynomials T(x) and G(x) one can produce a division T(x)/G(x). For example:

$T(x) = x^4+x^3+1$

and

$G(x) = x^{10}+x^9+x^7+x^5+x^4$

What is the remainder of G(x)/T(x)? Figure 2 shows a polynomial division example.

$$
\begin{array}{r|l}
 & x^6 + x^3 + x \\
\hline
x^4+x^3+1 & x^{10}+x^9+x^7+x^5+x^4 \\
 & \underline{x^{10}+x^9 \quad +x^6} \\
 & \quad x^7+x^6+x^5+x^4 \\
 & \quad \underline{x^7+x^6 \quad +x^3} \\
 & \quad\quad x^5+x^4+x^3 \\
 & \quad\quad \underline{x^5+x^4 \quad +x} \\
 & \quad\quad\quad x^3 + x
\end{array}
$$

Figure 2 – Polynomial division example [5].

Now if each power of x is represented by 0 or 1 it may be easier to represent the polynomial division. For example the number 1101 would represent the polynomial:

$$x^3 + x^2 + 1$$

As it would happen with any division method, the process will produce a quotient and a remainder. The remainder is key to the CRC calculation while the quotient may be safely discarded.

Follows a discussion on how to do a module 2 division using binary numbers. When executing the division it is possible to prove that additions and subtractions will be equivalent to a XOR operation. Part of the operations can also be simplified. Consider now that the divisor is ***G*** (this is also called generator in the case of CRC calculations), the dividend is ***M*** (this is the original message ***D*** appended with zeros), the quotient is ***Q*** and the remainder is ***R***. In this case a simple process can be described:

- get *n* most significant bits of ***M*** (*n* being the same number of digits of ***G***).

1. if the most significant digit is 1, write 1 to ***Q*** and make a XOR with ***G*** to find the partial remainder.

2. If the most significant bit is 0, put 0 in ***Q*** and keep the bits as the next partial remainder (in other words, make a XOR with 0000).
3. Shift the partial remainder to the left, discarding one bit (notice that this bit will always be 0...) and get the next bit (so we still have *n* bits to process). Repeat 1 and 2.

To clarify here is a numerical example (figure 3):

```
G = 1001, D = 101110, M = 101110000 , Q = ?, R = ?

1001 | 101110000
       1001        (first 4 digits of M are 1011, XOR with G...)
       00101 (add 1 to Q, get the 5th digit from M after XOR with C)
        0000 (0111, so the first is 0, XOR with 0000)
         1010      (get the 6th digit from M, XOR with G)
         1001      (XOR with G)
         00110     (get 7th digit, the most significant is 0...)
          0000
           1100    (8th digit, most significant bit is 1, XOR with G)
           1001
           01010   (get the last digit)
            1001   (XOR with C as the most significant bit is 1)
        R = 0011   (remainder, no more digits from M)

Q=101011 (just for checking, it will be discarded)
```

Figure 3 – A simple polynomial division using binary numbers.

There are two important aspects to pay attention to: the first has to do with the generator, for its first bit is *always* one. A consequence of that is that every time the XOR operation can be made (the partial remainder is big enough to be divided), then the resulting first bit will *always* be zero.

The second observation is related to the number of bits to be appended to the original message. The number of bits of the generator is one more bit then the number of bits to be appended. So for a 16 bits CRC value calculation the generator ***G*** has to be a 17 bits number.

4 – Bit manipulation

You should be comfortable with these operators, but let's recall how to use them:

& **bitwise AND**

| **bitwise OR**

^ **bitwise XOR**

<< **shift to the left**

>> **shift to the right**

These operators manipulate bit by bit. For this discussion only some of them might be useful. The operator **&** is used to find out if the most significant bit is either 0 or 1. The operators **>>** and **<<** are used to shift numbers to the write and to the left respectively (this is the equivalent to multiply or divide the number

by a power of 2). The last important operator is the ***XOR*** (^) for calculating the partial remainders.

Suppose we have a certain variable with value 0100 0001 (this is, by the way, the binary for the character "A"). In order to find out what the first bit is we can use a "mask", a special constant that masks everything except the bit we are interested in:

```
(0100 0001) & (1000 0000) = 0000 0000
```

if the character is instead 1100 0001:

```
(1100 0001) & (1000 0000) = 1000 0000
```

Notice that the mask makes all the bits equals zero except the most significant one. Using other mask values one can separate portions of bits of a certain variable.

The XOR is simple to understand. For a given pair of bits the resulting value is either 0 if the pair is of the same value or 1 if the pair is of different values:

```
(0100 0001) ^ (1000 1000) = 1100 1001
```

Finally the shift may be useful to do the following operation:

```
(0100 0001) << 1 = 1000 0010
```

or

```
(0100 0001) >> 2 = 0001 0000
```

It is easy to show that << 1 actually multiplied the number by 2, while >> 2 divided the number by four.

Notice that values may be lost to the right or to the left. Unfortunately the way the shift operators work depends on the variable type and may even depend on architecture. Generally speaking, if the variable is unsigned there is a guarantee that zeros will appear from the left (in case of doing shifts to the right). Otherwise the sign bit (which may happen to be one) is shifted. To avoid such problems one can use *unsigned chars*. The constants are unsigned, so they can be safely shifted. The word size (architecture dependent) may make the shift operators to work differently due to remaining bits on the left or on the right sides. To help to debug problems, a binary printout of the variables may be useful. The program below prints bits of any variable [1]:

```
// function to print anything in bynary

// From "A Book on C", Pohl, I. And Kelley, A., 1995
#include <limits.h>
void bit_print(int a)
{
        int i;
        int n = sizeof(int) * CHAR_BIT;
        //change n to int n=8; if wants only the first 8 bits
        int mask = 1 << (n-1);
        for(i=1;i<=n;++i){
                putchar(((a & mask) == 0) ? '0' : '1');
                a<<=1;
                if (i % CHAR_BIT == 0 && i < n)
                        putchar(' ');
        }
}
```

5 – Standard CRCs

In order to divide big decimal numbers there is no need to divide the whole number at once. Rather one gets number after number from the most significant digits until this partial result is big enough to be divided adequately. The partial remainder is then added to the numbers that are still available on the dividend. The partial remainders and the dividend digits are added respecting their *status quo*, or the position (a power of the basis) to where they belong to. The same idea may be applied in CRC. It is also possible to simplify the whole calculation of the CRC computing *character by character* (rather than bit by bit).

As mentioned before, the generator cannot be chosen randomly and after some research a few popular CRC generators where standardized. There are also a few extra characteristics that has to be defined by the standard, so the implementations can be made compatible. Table 1 presents a few standard CRCs.

The polynomial (the generator) is truncated, since the first bit is always one. For example 0x8005 has 16 zeros, but the actual polynomial has (implicitly) 17 bits.

All the examples so far used initial remainders equals to zero. In practice this may lead to awkward errors. Consider for instance a message that contains lots of zeros in the beginning. The remainder is zero until the first one is shifted and the first XOR operation can be done. This may lead to failure in detecting errors if bits are added or dropped (since this is not going to change the final remainder). To avoid that situation some implementations begin the calculations with a full remainder (all bits are set to one). A final XOR value can also improve the results for other errors that would not be detected otherwise.

The "check value" in the final row of table 1 (below) is the CRC value for a message containing '123456789'. This message is in ASCII, not integer (so "1" is actually 0x31, and "9" is 0x39 etc). This message can be used to double check that the implementation works correctly.

	CRC-CCITT	***CRC-16***	***CRC-32***
width	16 bits	16 bits	32 bits
polynomial	0x1021	0x8005	0x04C11DB7
Initial remainder	0xFFFF	0x0000	0xFFFFFFFF
Final XOR value	0x0000	0x0000	0xFFFFFFF
Reverse data	no	yes	yes
Reverse CRC result	no	yes	yes
Check value ("123456789")	0x29B1	0xBB3D	0xCBF43926

Table 1 – A few characteristics of common CRC standards [2].

The simple implementation with XOR operations is sometimes too slow to be used in practice. Notice that for a given generator there is a limited number of possible XOR results. For example, for an 8 bit CRC there will be around 2^8 possibilities. These partial results can be calculated in advance and stored in memory as part of the code. Using look-up tables, one can make the code work in real-time.

6 – Programming

A useful function that implements a simple CRC is available as part of the code for assignment 2. The code can be found under the name CRC_Simple.c. The function is:

```
#define GENERATOR 0x8005 //0x8005, generator for polynomial division
unsigned int CRCpolynomial(char *buffer){
      unsigned char i;
      unsigned int rem=0x0000;
      int bufsize=strlen((char*)buffer);
      while(bufsize--!=0){
            for(i=0x80;i!=0;i/=2){
                  if((rem&0x8000)!=0){
                        rem=rem<<1;
                        rem^=GENERATOR;
                  }
            else{
               rem=rem<<1;
               }
                  if((*buffer&i)!=0){
                     rem^=GENERATOR;
                  }
            }
            buffer++;
      }
      rem=rem&0xffff;
      return rem;
}
```

The function can be used directly by the client and server in assignment 2. The function reads a string (ending with \0), and computes and returns a 16 bits CRC according to the method described in figure 3.

There is a subtle difference between this function and the one used in the standardized protocols: there is no reverse of data, and no reverse of the final remainder. Therefore, the results of the computation of the string “123456789” is NOT going to be the one in table 1. Instead, the value is 0xFEE8.

References

1 – Kelley, A. and Pohl, I., *A book on C*, Addison-Wesley, 1995.

2 – Barr, M. (2000), *Easier said then done,* Online. Accessed August 2012. http://www.netrino.com/Connecting/2000-01/index.html.

3 – Ross, W. N. (1993), *A painless guide to CRC error detection algorithms*, Online. Accessed August 2012. http://www.ross.net/crc/download/crc_v3.txt

4 – Kurose, J. F. and Ross, K. W., Computer Networking, Addison-Wesley, 2003.

5 – Shay, W. A., Understanding Data Communication and Networks, ITP, 1999.

6 – Online CRC calculator http://www.zorc.breitbandkatze.de/crc.html, Accessed September 2013.

version September 2013

Exercises

1) Compute 16 bits CRCs for the following single string messages. Check the results *manually* and with the function given for the assignment 2 (CRC_simple.c), and with the online CRC calculator from reference (6). Remember to turn off any extra options from (6).

a) “A”

b) “B”

c) “C”

d) “1”

e) “2”

f) “3”

2) Compute larger messages using the CRC_simple.c code, and compare the results with CRC calculator from reference 6.

a) “123456789”

b) “PACKET 0 line 1 aaaaaaaaaaaaaaaaaaaa”

3) Change the generator to 1021 and repeat exercises 1) and 2).

4) Discuss what should be changed in the code (section 6 C code) in order to use a 32 bits CRC.