Error Detecting Codes

Simple parity checks are of only limited value. Errors in communication systems tend to occur in bursts which means that several bits are likely to be affected. Instead, a "Cyclic Redundancy Check" (CRC) (otherwise known as a "Polynomial Code") is use. There are two major attractions of this method

i. It is backed by some solid mathematical theory so its behaviour under different error conditions is predictable.
ii. It is cheap to implement in hardware.

The method relies on the mathematical theory of polynomials. The general form of a polynomial is

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_{a0}$$

This is said to be of "degree n" as this is the power of its highest term. The number a_r is called the coefficient of x^r; any of the coefficients except an may be zero. Note that a polynomial of degree *n* has *n+1* terms. To give a concrete example:

$$8x^6 + 3x^4 + x^2+1$$

Is a polynomial of degree 6. Note that if we insist that all the coefficients are between 0 and 9 and we set x = 10 then this can be interpreted as a decimal number. Likewise, if all the coefficients are 0 and I and we set x to 2 we get a binary number. It is this interpretation which allows polynomial theory to he applied to error detection. in binary messages

If we divide one polynomial by another the we get a quotient polynornial rind a remainder polynomial. If the remainder is zero (all coefficients zero) then we say that the division is exact. Polynomial division, is analogous to long division of decimal numbers. For example

```
      24
123 | 3071
     246
     611
     492
     119
```

So 3071/123 = 24 r 119
We can also say:

3071 = 24 x 123 + 119
If we subtract the remainder from each side we have:
3071 - 119 = 2952 = 24 x 123

There is a general rule here; if you subtract the remainder before doing the division the result will be an exact division (i.e. the remainder will be zero).

You can generalise long-division to polynomials, for example

$$
\begin{array}{r|l}
 & 8X^4 - 5X^2 + 6 \\
\hline
X^2+1 & 8X^6 + 3X^4 + X^2 + 1 \\
 & \underline{8X^6 + 8X^4} \\
 & \quad -5X^4 + X^2 \\
 & \quad \underline{-5X^4 - 5X^2} \\
 & \qquad\quad 6X^2 + 1 \\
 & \qquad\quad \underline{6X^2 + 6} \\
 & \qquad\qquad\quad -5
\end{array}
$$

Which says that

$8x^6 + 3x^4 + x^2 + 1 / x^2 + 1 = 8x^4 - 5x^2 + 6, \quad r -5$

or

$8x^6 + 3x^4 + x^2 + 1 = (8x^4 - 5x^2 + 6)(x^2 + 1) + -5$

Notice that: ***degree(quotient) = degree(dividend) - degree(divisor)***

and ***degree(remainder) < degree(divisor)***

The crucial step in the calculation of a CRC is division and inspection of a remainder. In a CRC check, as we will see, matters are so arranged that we *expect* the remainder to be 0 - if it is not 0 we conclude that an error has occurred. This is fine, but we must ask what chance there is that an error could occur and still the remainder is 0? Polynomial theory tells us the answer to this, question *and* helps us choose a divisor which will minimise the chance of getting a 0 remainder even when a n error has occurred

Recall that polynomials are relevant because - if all coefficients are. 0 or 1 and x = 2 - they represent binary numbers. Note, also, that any digital message exchanged between two computers can be regarded as a binary number and, hence, as a polynomial. Calculation of a CRC involves division of the "message" by some well-known "generator polynomial". Ordinarily division is quite expensive computationally. Fortunately, we can replace ordinary division by binary division mod 2 without invalidating the polynomial theoretic results.
The division "mod 2" is very simple (and is also simple to implement in hardware) - there are neither borrows and carries. The addition and subtraction tables mod 2 are identical:

+/-	**1**	**0**
1	0	1
0	1	0

This is also identical to Exclusive OR. In spite of this strange arithmetic, all the familiar properties about remainders still hold.

Here is an example of binary division mod 2. Do not try to impose analogies from normal arithmetic, simply regard this as a mechanical process. Note that the question "does x go into y" should be interpreted as "is the length of x <= that of y".

```
          1100100
1101 ⌈1011010110        -> 1100100 r 10
     1101
      1100
      1101
         1101
         1101
           10
```

The principle steps in the CRC algorithm are:

i. Chose a generator polynomial (i.e. a binary number) (G). (This has to be agreed in advance by both parties)
ii. Divide the message (M) by G and note the remainder R
iii. Subtract R from *M.* This will give a polynomial/binary number which is exactly divisible by G. The receiver checks this by doing the division and checking that the remainder really is zero.

The CRC algorithm in detail

For the transmitter

1. Choose a "Generator Polynomial" of degree r.
   Effectively, this means choose an r+1 bit pattern. The choice of this is crucial as we will see – in practice a few "standard" patterns are commonly used. Both sender and receiver must know what generator is being used. Note that the remainder after dividing by the generator will have degree < r - i.e. it will have r bits or fewer. This turns out to be important later.

2. Extend the message by appending r 0 bits to it.
   This ensures that the extended message is of degree at least r.

3. Divide the extended message by the generator (mod 2) and note the remainder. Effectively it is this remainder which is the checksum.

4. Subtract the remainder from the extended message.
   The result will be a polynomial (bit pattern) which will be exactly divisible by the generator.
   Since the remainder will have r bits (or fewer), and we extended the message with r 0 bits, the subtraction *only affects the additional zero bits;* the original message is unaltered. Although we have expressed these steps in terms of arithmetic operations, the result is simply the original message with the checksum appended. The example below should make this clear.

5. Transmit the extended message.

For the receiver:

1. Divide the received message by the generator (mod 2)
2. If the remainder is not zero

   then **ERROR**
   else remove the least significant r bits and retrieve the message

Example.

Message	11011100	(degree 7, 8 bits)
Generator	1100	(degree 3, 4 bits)
Extended Message	11011100**000**	(degree 10, 11 bits),

Divide extended message by generator (step 3)

```
      10010111
1100 ) 11011100000      so the remainder is 100 (degree 2, 3 bits)
       1110
        1110
        1100
          1000
          1100
           1000
           1100
            1000
            1100
             100
```

Subtract remainder from extended message (step 4)

```
11011100000
     -100
-----------
11011100100      11011100100 is transmitted
```

Receiver calculates:

```
      10010111
1100 ) 11011100100      so the remainder is 0 and the CRC succeeds.
       1110
        1110
        1100
          1001
          1100
           1010
           1100
            1100
            1100
            0000
```

The original message is now retrieved simply by discarding the 3 rightmost bits.

The effect of errors

In practice, it appears that errors occur in bursts; a group of (say) 10 bits will mostly be affected then there will be, perhaps, several thousand bits before another error occurs. The power of the CRC check lies in its ability to detect these kinds or error burst. Mathematically, a burst of errors is equivalent to adding (mod 2) some random number to the transmitted message.

Suppose the transmitted message is *M* and the error added is e, then the received message will be *M′ = M+e.* The check the receiver will perform is:

$$\frac{(M+e)}{G} = \frac{M}{G} + \frac{e}{G}$$

If this happens to give a zero remainder then the CRC check will have failed to detect the error. Since we already know that M is divisible by G, we can concentrate on the second term - e*lG.* If *elG* gives a zero remainder we are in trouble; if not, we are OK.

Consider an error burst which affects one bit, leaves the next two bits intact, then affects the fourth bit. This is equivalent to adding 1001 to the message. However, it is also equivalent to adding, 10010, 100100, 1001000 etc. depending on exactly where in the message the error burst occurred. It is clear from this example that e will normally have many trailing zeros. We can infer from this the first and most simple rule about how *G* should be chosen; it must not have trailing zeros (otherwise it may well divide a exactly). Our generator is clearly a bad choice

Assuming *G* ends in a 1 it corresponds to a polynomial of the form $x^i + ... + 1$. If the error burst affects only *one* bit it will have the form x^n.

Thus $\frac{e}{G} = \frac{x^n}{(x^i + ... + 1)}$ which cannot be zero. Hence all 1-bit errors will be detected

Suppose also that we choose *G* so that it has a factor (x + 1). e/G will only give a zero remainder if we can cancel out this factor - i.e. if (x+l) is also a factor of *e.* It is easy to show that e cannot have (x+l) as a factor if it represents an error burst affecting an odd number of bits:

Recall that the simple parity check performs poorly with error bursts since each successive bit in error tends to mask the effect of the *previous* bit.

a) An error burst affecting an odd number of bits can be represented as a polynomial E(x) which has an odd number of non-zero coefficients. Given that we are using mod 2 arithmetic, it is clear that E(1) = 1 (remember that 1 + 1 = 0).

b) If E(x) has (x+l) as a factor we can write E(x) = (x+l)F(x) so E(1) = (1+1)F(1) = 0

a) and b) are contradictory hence we can see that *E(x)* cannot have (x+l) as a factor so the error will be detected.
We now know that if G ends in 1 and has (x+1) as a factor it will detect all 1-bit errors and all errors affecting an odd
number of bits. By exploiting other features of the theory of polynomials we can put further constraints on G so as to

increase the effectiveness of the CRC. Careful study of what is required has led to the choice of a few standard polynomials, for example:

CRC-16 $x^{16}+x^{15}+x^2+1$
CRC-CCITT $x^{16}+x^{12}+x^5+1$

Both give 16-bit checksums which will detect:

- All 1 and 2 bit errors
- All error bursts of up to 16 bits in length
- All bursts affecting an odd number of bits
- 99.997% of 17 bit error bursts
- 99.998% of 18 and longer bursts