Dealing with Bit level Errors

Peter Rounce (P.Rounce@cs.ucl.ac.uk)
Room 6.18

Notes Courtesy of Graham Knight

Errors and error rates

- Bit Error Rate (BER):
  - e.g. 10^{-12} is one error in 10^{12} bits (on average)
  - For 64Kbps, a BER of 10^{-12} gives one bit error every 6 months.
- **Signalled error rate (SER)**:
  - detected errors
- **Residual error rate (RER)**:
  - errors that are not detected or corrected
- Error correction:
  - if either SER/RER is unacceptable
- Error detection:
  - if RER is unacceptable

06/02/2006 2011-06 2

We can only conclude that a service has been guilty of error if we know the QoS it was supposed to provide. If we are using a CL service which gives no guarantees about sequenced delivery then we should not complain if it delivers out of sequence. However, there are a couple of situations that may be considered errors no matter what service we are considering:

• delivery of a message which has been corrupted in some way, i.e. the bits received are not the same as the bits sent.

• non-delivery of a message that was submitted.

We will begin by concentrating on these two error types. No service can offer a zero error rate. However, it is possible for the error rate to be effectively zero (negligible). For example, if a 64 Kb/s line has a BER of 10^{-12} there will be one bit in error in every 6 months of continuous operation. In most circumstances this is not worth worrying about; other associated bits of equipment are breaking much more frequently than that.

Two error rates are important:

• the signalled error rate concerns errors that are detected and reported by the underlying service.

• the residual error rate concerns errors that are missed by the underlying service.

Redundancy

- Here is a **srelling** mistake
- Not all combinations of letters make legal words:
  - if they did how would we detect error?
- Redundancy:
  - information that is not essential
  - helps in detecting errors
- To detect/correct errors, we need redundancy:
  - how much do we need?
  - what is the cost?

06/02/2006 2011-06 3

It is obvious that there is a **srelling** mistake in this sentence. It is obvious because the set of all possible words that can be made from English letters can be partitioned into two sets; English words and non-English words. **Srelling** belongs to the latter set as do **dfede**, **msndasad**, and **qjykfda**. In fact, vastly more "illegal" words can be made than "legal" ones. Suppose we somehow re-organised English spelling so that all combinations of letters spelled legal words. We would not now be able to tell whether a word had been mangled by a transmission service; the result would always be another legal word. We may still be able to detect the error from the context. However, this is really equivalent to saying the set of all possible sentences that can be made from English words can be partitioned into two sets; English sentences and non-English sentences.

Another way of putting this is to say that our spelling system has **redundancy**. All those illegal combinations are wasted as far as the business of encoding information is concerned. We saw this earlier when we looked at entropy.

If we adopted a more efficient encoding so that all the redundancy was removed, it would make the construction of cross-words trivial. Any combination of letters could be entered and all would be valid words. However, solving the puzzle would be much harder since a much larger range of possible solutions would fit. On the other hand, if English were much more redundant, construction of cross-words would be almost impossible.

What is true for spelling is also true for the way data must be encoded for data transmission. If we are to detect errors then there must be some redundancy. This means that error control has a cost in terms of throughput; the capacity of a channel is effectively reduced (with respect to the amount of *information* transmitted) since some of the bits it carries are redundant. We need to understand how much redundancy is needed in order to be able to detect errors.

Error detection

- Messages: size *m*
- **Codewords**: size *c*
  - *c* = *m* + *r*
  - *r* error control bits
- Example – ASCII (8-bit):
  - 7-bits data, 1-bit even parity
- **Hamming distance**
- To detect *h*-bit errors: **Hamming distance > *h***

parity bit

```
A  0 100 0001
B  0 100 0010
C  1 100 0011
```

Min. Hamming distance = 2

```
1-bit error:  0100 0101  ×
              0000 0010  ×
              1000 0011  ×

2-bit errors: 1100 0011  ?
              0110 0011  ?
              0100 0001  ?
```

06/02/2006 2011-06 4

To make the problem manageable we will assume that information is to be transferred in fixed-size chunks of *m* bits (*m* for "message"). Thus there are 2^m different possible messages that might be sent. Messages are encoded for transmission as *c* bits where $c = m + r$. The additional *r* bits constitute the required redundancy. Note that it is not necessarily the case that we can classify each bit as "information" or "redundant" any more than we can say which letters in an English word are redundant. We call each of the *c*-bit objects a **codeword**. There are 2^c possible codewords but only 2^m of these are legal codewords – the ones corresponding to the messages. Bit errors alter bits in codewords. When this happens we would like the result to be an illegal codeword. If the result is a legal codeword then the error will be missed. For example, suppose we construct our codewords on the basis of applying even parity to 7-bit ASCII. This gives the following codewords:

```
A  0 1 0 0  0 0 0 1
B  0 1 0 0  0 0 1 0
C  1 1 0 0  0 0 1 1
```

If we alter one bit in any of these codewords the result is an illegal codeword since the parity would be wrong. However, if we alter two bits then the result will be legal. We can change "A" to "B" in this way. With this encoding, all legal codewords are a least two "bit changes" apart. The number of bits in which two codewords differ is called the **Hamming distance**. In the 8-bit ASCII code above, there are no two legal codewords with a Hamming distance of 1. We can say that the *minimum* Hamming distance is 2.

To detect all *h*-bit errors, minimum Hamming distance > *h*

8-bit ASCII has 1 redundant bit in 8. This is 12.5% redundancy.

Single-bit parity: residual error rate

- Example:
  - Transmission of 8-bit bytes
  - raw BER of channel = 10^{-4}
  - P(A bit is in error) = p = 10^{-4}, P(Bit not in error) = 1-p
- No parity
  - P(No bit errors in a byte) = $(1-p)^8$
  - P(At least one bit error in byte) = $1 - (1-p)^8 = 8\text{x}10^{-4}$
- Add a parity bit (nine bits in all)
  - P(Exactly two bits in error) = ${}^9C_2 p^2(1-p)^7 = 3.6\times10^{-7}$
- Parity-bit gives 3 orders of magnitude improvement(!)

06/02/2006 2011-06 5

With a single parity bit, r, for our codewords of length c, the value of r is such that it turns the number of 1 bits in the whole codeword to be odd (even). The pre-assigned codewords are separated by a Hamming distance of 2 or more. To see how the addition of a parity bit can improve error performance, consider the following example. A common choice of c is eight. Suppose that the raw BER, p, of the channel is measured to be $p = 10^{-4}$. Then:

P{single bit error} $= p$

P{no error in single bit} $= (1-p)$

P{no single bit error in 8 bits} $= (1-p)^8$

P{undetected error in 8-bits} $= 1 - (1-p)^8 = 8.00\times10^{-4}$

So the residual error rate is 7.9×10^{-4}. With the addition of the parity check, we can detect single bit errors but we will fail if there are two errors. We will also fail if there are 4, 6 or 8 errors but we will ignore these are the probabilities are tiny.

P{exactly 2 errors in 9 bits} $= {}^9C_2 p^2(1-p)^7$ (Binomial distribution)

$= 3.6\times10^{-7}$

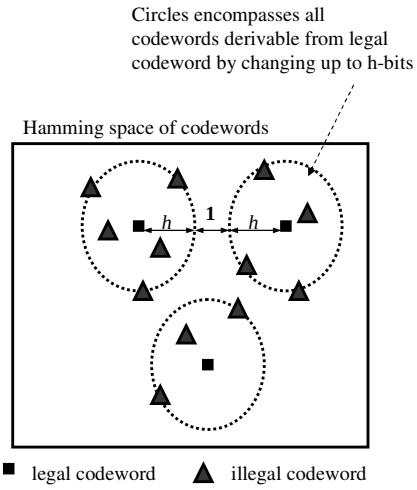
[Essentially the maths is the number of combinations of ways in which 2 bits can be chosen from 9 (${}^9C_2$) multiplied by the probability of each occurring $p^2(1-p)^7$.

As can be seen, the addition of a parity bit has reduced the undetected error rate by three orders of magnitude.

Single parity bits are common in asynchronous, character/byte-oriented transmission. Where synchronous transmission is used, other methods are used (we shall look at one of these in detail later).

Error correction

- Hamming distance:
  - *h*-bit error: can it be corrected?
  - Yes, if enough distance between codewords
  - what is "enough"?
- To correct *h*-bit errors: **Hamming distance ≥ 2*h* + 1**



Codewords **2-bit error** **3-bit error**

```
00000 00000    00000 10001    00000 10001
00000 11111
11111 00000
11111 11111
```

Min H.D.of these codewords = 5

06/02/2006 2011-06 6

Encodings that allow correction in addition to detection require more redundancy and a greater Hamming distance. Suppose we want to be able to correct all errors of *h* bits or fewer. We must ensure that the set of all possible *illegal* codewords that we can generate from a legal codeword by changing *h* bits or fewer does not intersect the similar set generated from *any other* legal codeword. If we receive one of these illegal codewords, (i.e. one generated from a legal code-word by altering *h* bits or fewer) we will always be able to tell from which legal codeword it was generated. We can think of each legal codeword as being surrounded by a region of illegal codewords which can be generated from it by altering *h* bits or fewer. None of these regions must intersect. We can state that:

To correct all *h*-bit errors, minimum Hamming distance ≥ 2*h* + 1

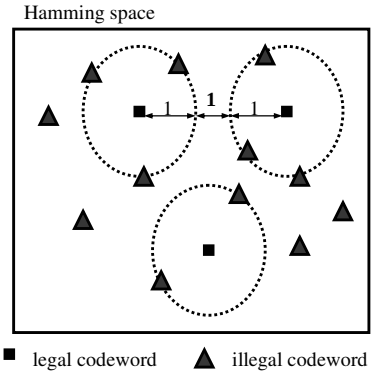
For example, to correct all 1-bit errors requires a minimum Hamming distance of 3. So 8-bit ASCII has no error correcting properties. Suppose we have a system with just four codewords:

```
0000000000  0000011111  1111100000  1111111111
```

This has a minimum Hamming distance of 5, so it should be able to correct all 2-bit errors. The nearest legal codeword is the result of seeing which are the fewest number of bits we need to alter to get to a legal codeword.

How much Redundancy

- Suppose we want to correct all 1-bit errors
- $c = m + r$
- Each circle includes $1+c$ codewords
- There are 2^m circles
- There are 2^c codewords
- $(1+c)2^m \leq 2^c \Rightarrow 1+m+r \leq 2^r$
- E.g. m = 7, r = 4



06/02/2006 2011-06 7

How much redundancy would be required to *correct* all 1 bit errors? We have 2^m messages and hence 2^m clouds. Since codewords are c bits long we can generate c illegal codewords from each by changing 1 bit of a legal codeword. So each region accounts for c+1 codewords (c illegal plus one legal). So, the total number of codewords in the clouds is $2^m(c+1)$. This must be less than or equal to the absolute total number of codewords (there may be some illegal codewords not in any regions) so:

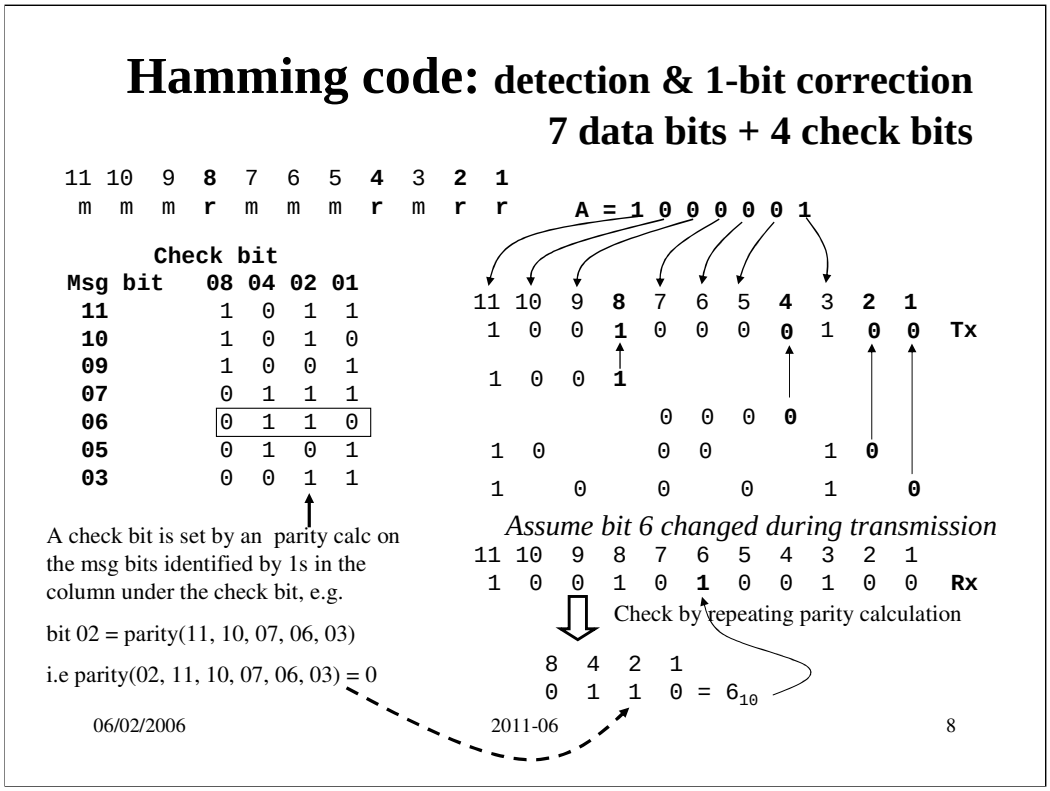
$$2^m(c+1) \leq 2^c \quad \Rightarrow \quad 2^m(m+r+1) \leq 2^{m+r}$$

(since $c = m+r$)

$$\Rightarrow m+r+1 \leq 2^r$$

m = 7 (for ASCII) and using some values for r, we find the inequality is satisfied at r = 4.

This does not tell us how to construct a suitable code of course. All we know is that it must have 11 bits and that the minimum Hamming distance must be 3 or more. Fortunately Mr Hamming invented a way of doing the job …



Hamming not only derived a limit on how many check bits would be needed, he devised a code that operated at that limit – the **Hamming Code**. Let us look at a Hamming Code for 7-bit ASCII. This must have 4 check bits. The check bits are placed in the power-of-2 positions; 1, 2, 4 and 8. The check bits act as 4 independent parity bits for combinations of the message bits. To see which message bit is checked by which check bit write the number of the message bits as binary numbers. This indicates that check bit 8 will check message bits 11, 10 and 9; i.e. bit 8 is chosen to give even (odd) parity to bits 11, 10, 9, and 8.

The table also shows which check bits will be affected by an error in a given message bit. For example, if bit 9 is altered, check bits 8 and 1 will yield the wrong parity, 4 and 2 will be unaffected. Therefore, if we find the checks for 8 and 1 fail whilst those for 4 and 2 succeed it must be bit 9 that is in error and we can correct it. Note also that if it is one of the check bits that is altered the other checks will be unaffected. The error correcting algorithm is:

"Write down a 1 under the check bits that fail and a 0 under those that succeed. Interpreting the result as a binary number gives the number of the bit that is in error."

For example, ASCII "A" (100 0001), even parity, would be encoded as:

```
11 10  9  8  7  6  5  4  3  2  1
 1  0  0  1  0  0  0  0  1  0  0
```

If this is received as:

```
11 10  9  8  7  6  5  4  3  2  1
 1  0  0  1  0  1  0  0  1  0  0
```

Then we write:

```
 8  4  2  1
 0  1  1  0
```

since the checks for bits 4 and 2 fail. This tells us that bit 6 should be corrected. This procedure can be implemented very cheaply in hardware.

Error detection codes for packets

- Parity check - limited use:
  - large frames?
  - burst errors?
- **Polynomial codes**:
  - cyclic redundancy check (CRC)
  - frame check sequence (FCS)
- Based on binary arithmetic mod 2

```
          24
   123|3071     So: 3071/123 = 24 r 119
      246       or: 3071 – 119 = (24×123)
      611
      492
      119
```

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

So: $8x^6+3x^4+x^2+1 \,/\, x^2+1 = 8x^4-5x^2+6$, r -5
or: $8x^6+3x^4+x^2+1 - (-5) = (8x^4-5x^2+6)(x^2+1)$

$$\begin{array}{r|l} 1.10^2+2.10+3 & 3.10^3+\quad 7.10+1 \\ & \underline{3.10^3+6.10^2+9.10} \\ & \quad -6.10^2-2.10+1 \\ & \quad \underline{-6.10^2-12.10-18} \\ & \qquad\quad 10.10^2+19 \end{array}$$

3071/123 = 24 r 119

310 - 6

Simple parity checks are of only limited value. Errors in communication systems tend to occur in bursts which means that several bits in close proximity are likely to be affected. Instead, a **cyclic redundancy check (CRC)** (otherwise known as a **polynomial code**) is used. There are two major attractions of this method:

1. It is backed by some solid mathematical theory so its behaviour under different error conditions is predictable.

2. It is cheap to implement in hardware.

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

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

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 a_n 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 1 and we set x to 2 we get a binary number. It is this interpretation which allows polynomial theory to be applied to error detection in binary messages. If we divide one polynomial by another the we get a **quotient polynomial** and a **remainder polynomial**. If the remainder is zero (all coefficients zero) the we say that the division is exact. Polynomial division is analogous to long division of decimal numbers.

There is a general rule; 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.

Polynomial codes [1]

- Use remainder (*R*) as checkbits
- Sender "subtracts" remainder from message (*M*)
- Receiver re-computes remainder:
  - If non-zero ⇒ error
- Must chose "good" **generator polynomial (*G*)**
- Mod 2 arithmetic - simple in hardware

+	1	0
1	0	1
0	1	0

-	1	0
1	0	1
0	1	0

Arithmetic Mod 2

```
          1100100
1101|1011010110   ⇒ 1100100 r 10
    1101↓
     1100
     1101↓
        1101
        1101↓
          10
```

- *R* always fewer bits than *G*

06/02/2006 2011-06 10

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, 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 an error has occurred.

Note 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 this 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) – the question "does *x* go into *y*" should be interpreted as "is the length of *x* ≤ that of *y*". There are neither borrows or carries. The addition and subtraction tables mod 2 are identical, and are the same as the XOR function.

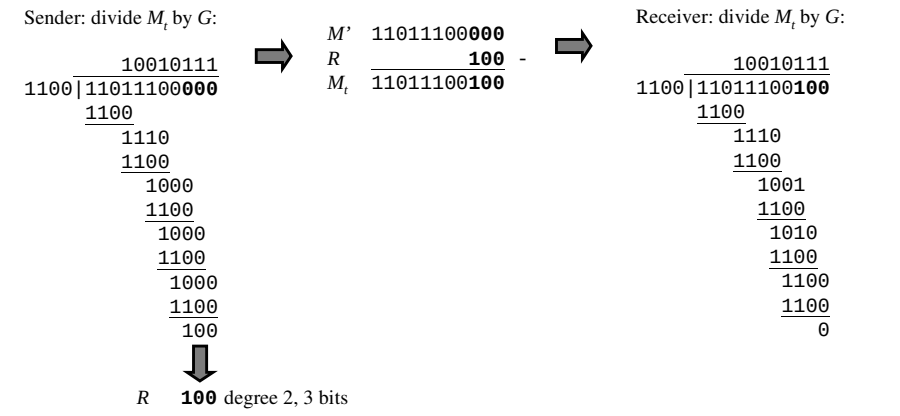
The principle steps in the CRC algorithm are:

• Chose a generator polynomial (i.e. a binary number) (*G*). (This has to be agreed in advance by sender and receiver).

• Divide the message (*M*) by *G* and note the remainder *R*.

• 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.

Polynomial codes [2]



For the transmitter:

1. **Choose a Generator Polynomial (*G*) of degree *r*.** Effectively, this means choose a (*r*+1) bit pattern. The choice of this is crucial – 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 (*R*).** 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.

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 an error has occurred** in the transmission, else remove the least significant *r* bits and retrieve the original message.

Polynomial codes [2]

M_t transmitted message
M_E received message including error
E error

$$M_E = M_t + E$$

$$\frac{M_E}{G} = \frac{M_t + E}{G}$$

$$= \frac{M_t}{G} + \frac{E}{G}$$

For this to give a non-zero remainder
E/G must give a non-zero remainder

A 1-bit error => $E = x^n$

Suppose G ends in a 1

$$\frac{E}{G} = \frac{x^n}{x^k + \cdots + 1}$$

will always have a remainder.
All 1-bit errors will be detected.

06/02/2006 2011-06 12

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 of **burst errors**. 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. Mathematically, a burst of errors is equivalent to adding (mod 2) some random number to the transmitted message. Suppose the transmitted message is M_t and the error added is E, then the received message will be $M_e = M_t + E$. The check the receiver will perform is:

$$\frac{(M_t + E)}{G} = \frac{M_t}{G} + \frac{E}{G}$$

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_t is divisible by G, we can concentrate on the second term, E/G. If E/G 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 E exactly). Our generator (1100) 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 this means E 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.

Polynomial codes [3]

1. Assume $E(x)$ has odd number of non-zero coeffs. This means $E(1) = 1$

2. Choose G to have $(x + 1)$ as a factor then E/G will give 0 remainder only if $E(x) = (x + 1)F(x)$
Put x=1 in this gives $E(1) = 0$

1 and 2 are contradictory!
Therefore we do not get a Zero remainder and the Error is detected

So, $G(x) = (x + 1)(x^k + \ldots + 1)$ will detect:
- all single bit errors
- all odd-number bit errors

Lots of other properties possible …

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

06/02/2006 2011-06 13

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 $(x+1)$ is also a factor of E. It is easy to show that e cannot have $(x+1)$ as a factor if it represents an error burst affecting an odd number of bits:

1. 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 $1 + 1 = 0$).

2. if $E(x)$ has $(x+1)$ as a factor we can write $E(x) = (x+1)F(x)$ so $E(1) = (1+1)F(1) = 0$

It is clear that 1. and 2. are contradictory hence we can see that $E(x)$ cannot have $(x+1)$ as a factor so the error will be detected.

So, 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

Other error correction techniques

- **FEC**:
  - correct error(s) at receiver
  - computationally expensive
  - additional redundancy
  - may be useful for real-time communication or for channels with high data rates and high delays
- Convolutional codes:
  - Viterbi
- Reed-Solomon

06/02/2006 2011-06 14

When errors are detected at the receiver, the receiver can arrange for the sender to re-transmit the erroneous message using **automatic repeat request (ARQ)** protocols (we look at these later).

In some cases, the use of error control with ARQ protocols is not sufficient, for example due to real-time constraints, where there is a large end-to-end delay, or large **bandwidth-delay product**. In such cases, we must use other methods, such as **forward error correction (FEC)** – correction of errors at the receiver. The Hamming code we examined earlier is not suitable for large blocks of data and other mechanism such as convolutional codes must be used. (We do not examine these in detail in this course.)