1 GSW… Error Detection

One of the key requirements of just about any form of data transmission is that the data received by the receiver is the same as the data that was sent by the transmitter. No errors, no additional bits, no missing bits. In some cases the results of any bit errors that do occur are merely irritating (the copy of a downloaded document has a spelling mistake) and in other cases potentially disastrous (the decimal place ends up in the wrong place when we 'wire' money around the world, or a rocket sets off in the wrong direction).

Unfortunately, the world being how it is[1], there is always a finite possibility that the data sent over any physical link picks up an error on the way. All we can do is try and detect these errors in an effective and efficient way, so that if the data does arrive with an error in it, and we're worried by this error, we can ask for the correct data to be resent.

1.1 Evaluating Error Detection Schemes

The only possible way to detect errors is to transmit some *redundant information* that can be used to detect whether there was an error or not. For example, suppose I wanted to transmit the data '10010110', and the only thing the receiver knows in advance is that the packet is eight bits long. As far as the receiver is concerned, I could be sending any possible pattern of eight bits.

If any of these eight bits is corrupted and received in error, what results is another valid message, just not the one I sent. There is no way the receiver can tell that an error has occurred: it receives what appears to be a valid packet. The only way to detect errors is if the receiver can tell the difference between valid packets and invalid packets, and if it's possible to receive an invalid packet, there must be some redundant information in the packet.

It is also impossible to design an error detection scheme that can detect all possible errors. Assuming I have any information to send at all, there must be at least two different messages I can send (otherwise there's no point in communicating at all, the receiver would already know the only possible data message). If there is more than one valid message, then it must be possible for some pattern of errors to transform one valid message into another valid message. Such an error could not be detected at the receiver.

All we can do is try and minimise the number of undetected errors, and reduce the amount of additional redundant information we have to transmit with the data. An *effective* error detection scheme is one that detects almost all possible patterns of errors in the transmitted message; an *efficient* error detection scheme is one that does not require many additional bits to be transmitted with the original data.

To measure the effectiveness, we'll consider the probability of an undetected error. Detected errors aren't that serious: you can always ask for the data to be retransmitted. Even a few false positives (correct packets that the receiver thinks have errors in them) can be tolerated; it's the undetected errors that can cause chaos. To measure the efficiency, we'll consider the ratio of the number of useful information bits to the total number of transmitted bits.

---

[1] The world is full of Gaussian noise sources; and the Gaussian distribution, in theory at least, extends to plus and minus infinity. No matter what the signal to noise ratio is, if the noise is Gaussian, there is always a finite chance that there will be enough noise at the critical time to cause a bit error.

The three most common techniques for error detection are the parity check, the checksum, and the cyclic redundancy check. In this chapter we'll compare the schemes, and derive approximate expressions for their efficiency and effectiveness.

1.2 The Parity Check

The most efficient error detection technique is known as a *parity check*, as this scheme requires just one bit to be added to the end of each frame[2]. In *even parity* this additional bit is chosen to ensure that the total number of '1' bits transmitted is even; in *odd parity* the bit is chosen to ensure that the total number of '1' bits transmitted is odd. The efficiency of this simple scheme is then:

$$Efficiency_{parity} = \frac{L}{L+1} \tag{0.1}$$

where L is the number of information bits in the frame. While undoubtedly efficient, the effectiveness of this scheme leaves a lot to be desired; for example this scheme is not able to detect an even numbers of errors in the transmission. In real life, bursts of errors caused by noise spikes lasting multiple bit periods are quite common, so just making the assumption, as we are about to do, that errors occur independently is not always true; in fact bursts of errors can (and commonly do) occur that can be any number of bits long, so the resultant undetected error rate can be much higher than the following calculation suggests.

1.2.1 The Effectiveness of a Single Parity Bit

Consider a frame of L bits, protected by a single parity bit. If we assume that all errors are independent, and that the probability of any bit being received in error is p, then we can calculate the probability that there is one bit error in the resultant received frame of bits[3]:

$$\text{prob of one error} = p_e(1) = (L+1)\,p\,(1-p)^L \tag{0.2}$$

All such errors will be detected by the parity bit, as a single bit error will change the number of '1' bits in a frame from odd to even, or vice versa. The probability that there are two errors in the frame is:

$$\text{prob of two errors} = p_e(2) = \frac{(L+1)L}{2}\,p^2\,(1-p)^{L-1} \tag{0.3}$$

---

[2] I'm using the word 'frame' here rather than 'packet' since parity (and the CRC later on) are mostly used at the data-link layer. Checksums, on the other hand, are usually used at higher layers in the protocol stack, so I'll use the word 'packet' when I talk about those.

[3] For those not familiar with statistics, this is the probability that there is a bit error in one bit p times the probability that there isn't a bit error in any other bit $(1 - p)^L$, times the number of ways that this can happen: there are $(L + 1)$ bits in the frame including the parity bit.

For two bits in error, there are $L(L + 1) / 2$ possible combinations of two bits in the frame, and two bit errors in the data and the parity bit always results in an undetected error.

and all such errors are not detected by a single parity bit. For example, transmit '00000' and receive '00110', and a single parity bit would not detect the error: the number of '1' bits is even in both cases. Similarly, there is the chance that there would be three errors in the frame (which would be detected), or four (which wouldn't), and so on, however if the link is to have any chance of working with a parity check with independent errors, then we can usually assume that the probability of a bit error p is small, and neglect the possibility that more than three bit errors occur in any one frame.

Therefore, the probability that a frame arrives with undetected errors is approximately:

$$\text{prob of undetected error} = p_{ue} \approx \frac{(L+1)L}{2} p^2 (1-p)^{L-1} \tag{0.4}$$

1.2.2 The Effectiveness of Multiple Parity Bits

For large messages with a significant error probability, the probability of the frame arriving with two or more errors can be quite large. However, we can still use parity checking as a simple form of error detection in this case, by dividing up the frames into blocks of B bits, and adding a parity bit to each block. If there are a total of L data bits to send, then approximately L / B blocks will be required, and with one additional parity bit per block, the total number of bits transmitted will be $(L + L / B)$.

As before, the probability of any block arriving with a detected error (neglecting the possibility of three or more errors per block) is:

$$\text{prob of one error} = p_e(1) = (B+1)\, p\, (1-p)^B \tag{0.5}$$

and the probability of the entire message arriving with no detected errors is the probability that every one of the blocks arrive with no detected errors:

$$\text{prob of no detected errors} = \left(1 - p_e(1)\right)^{L/B} \tag{0.6}$$

There are only two ways in which an entire message can arrive with no detected errors: either it arrives entirely correctly, or it arrives with one or more undetected errors. The probability that the entire message arrives correctly is just the probability that none of the bits arrives with an error:

$$\text{prob of no errors in entire frame} = \left(1-p\right)^{\left(L+L/B\right)} \tag{0.7}$$

since a total of $(L+L/B)$ bits must be transmitted. Therefore, the probability of undetected errors, neglecting the probability of more than two errors per block, is:

$$\begin{aligned} p_{ue} &= \text{prob of no detected errors} - \text{prob of no errors} \\ &= \left(1 - \left(B+1\right) p\, (1-p)^B\right)^{L/B} - \left(1-p\right)^{\left(L+L/B\right)} \end{aligned} \tag{0.8}$$

and to get this level of undetected errors in the whole message, requires L / B parity bits. (Note this formula is only accurate when the probability of errors is very small, and we can neglect the probability of three or more errors in one block.)

There is a trade-off between the number of parity bits used, and the probability of undetected errors in an entire frame: with more parity bits the probability of an undetected error in the entire message goes down (and hence the effectiveness of the scheme increases), although more parity bits need to be added (and hence the efficiency of the scheme decreases).

1.2.3 Example of Parity Checking

As an example of this trade-off: suppose that we have a link with a bit error rate of 0.1% (so that $p = 0.001$) and we have a total message size to send of $L = 720$ bits. What is the probability of undetected errors in the message?

Plugging the numbers in to the equation above gives, for various values of block length B:

Bits per Block	Number of Blocks	Total Number of Bits Sent	Efficiency	Probability of Undetected Error per Block	Probability of Message with Undetected Errors
1	720	1440	1/2	1 e-6	0.17 e-3
2	360	1080	2/3	3 e-6	0.37 e-3
3	240	960	3/4	6 e-6	0.55 e-3
4	180	900	4/5	10 e-6	0.73 e-3
5	144	864	5/6	15 e-6	0.91 e-3
6	120	840	6/7	21 e-6	1.09 e-3

The best we can do is a probability of getting the message through with no undetected errors a proportion 0.17e-3 of the time (0.017%), and that requires us to send twice the number of bits as are in the original message. This really isn't very good. It's worth emphasising again that the analysis above assumes that all errors are independent. In real life, this is often not the case, and errors occur in bursts. Parity checks are particularly bad at detecting bursts of errors: as we've seen, two (or four, or six) errors in the same chunk and parity cannot detect anything is wrong.

Parity checking is used on the serial ports of PCs, and for memory checking in DRAMs, but the only information it can usefully give is that the link is prone to error. If any frames arrive with invalid parity, then the serial connection / memory is suspect, and no information received from that source should be trusted.

For other applications, we're going to need a different technique, one that can provide a much greater level of protection, doesn't require so many additional bits to be transmitted, and can ideally provide a greater level of protection against bursts of errors. Fortunately, two such techniques exist, and are widely used.

1.3 Checksums

A much more common error detection technique in protocol headers is the checksum. In their simplest form, checksums consists of treating all the bytes of data in a packet as numbers, adding them all up, taking the result modulo some number 2^N (in other words taking the least significant N bits of the answer), and transmitting this result at the end of the packet. While this technique can detect all single bit errors and most error bursts, there are lots of combinations of errors that it cannot detect. Nevertheless, checksums are widely used in the

Internet, particularly in higher layers of the protocol (the network layer and above), since they are so easy to calculate in software.

For example, consider the data 0xADD16E, and an eight-bit checksum. Expressing the data to be transmitted as three eight-bit binary numbers and adding them up gives a checksum of 0xEC:

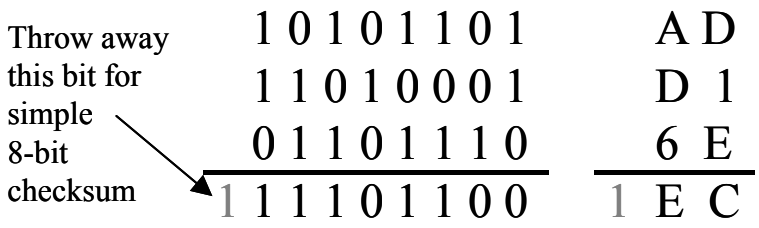

Figure 1 - Generating a Simple 8-bit Checksum

1.3.1 Effectiveness of Checksums

It is quite easy to extend the result we derived for parity bits to produce an approximate expression for the probability of an undetected error when using checksums. Compare the use of the checksum, to the use of one parity bit for each column of the data. If we consider patterns of two errors only in the packet, the only possibilities for undetected errors if parity is used with each column are if two bits are received in error in the same column. If checksums are used, then again both errors must be in the same column, but now one of them must be a zero, and the other a one, otherwise the next bit in the checksum will be wrong. For example, consider the patterns of bits below:

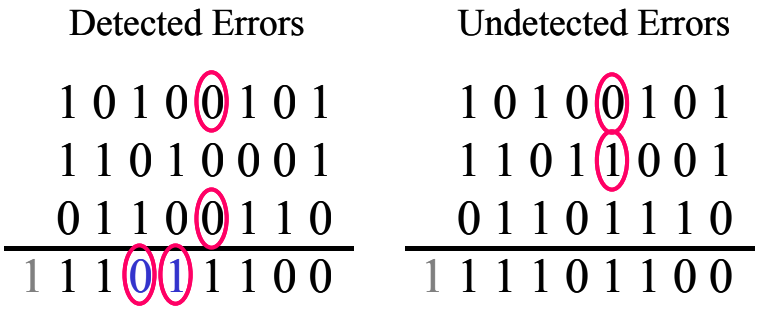

Figure 2 - Detected and Undetected Errors with Checksums

In the first example, the two bits received in error are both '1's that have become zeros. If the packet were being protected using a single parity bit per column of data, this error would not be detected, as the number of ones in this column has not changed. However, with a checksum as shown the error is detected, as with the correct data, there would have been a '1' to carry over to the next column, and this makes the sum different.

If we assume that there are only two bits that are received in error in one column, then we can make a simple estimate of the error-detection capabilities of the checksum. First, we have to note that there are four possible ways to get two bit errors in a column of bits:

First Error	*Second Error*	*Detected by Checksum?*
'1' becomes '0'	'1' becomes '0'	Yes

'1' becomes '0'	'0' becomes '1'	No
'0' becomes '1'	'1' becomes '0'	No
'0' becomes '1'	'0' becomes '1'	Yes

and a checksum will detect two of them. Again, we'll make the assumption that we can neglect the probability of more than three errors in any one column.

You might have noticed that using the simple scheme described above, any two errors in the most significant bit of the data will produce an undetected error, however these cases can be avoided by adding any carry bits in the sum back into the checksum, rather than ignoring them. For example, in the case above, we'd get:

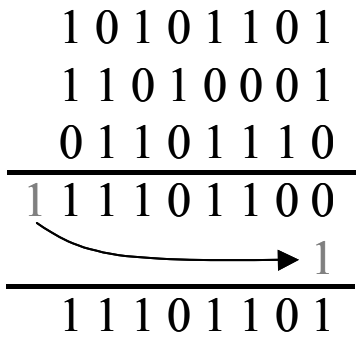

Figure 3 - Using the Carry Bits in the Checksum

Then; if there are two errors in a column, the probability of both of them being the same (either both zeros or both ones) is 50% (assuming each bit in the data is equally likely to be a one or a zero, and there is no correlation in the data). These patterns will be detected by checksums, but not by parity checks. Therefore, the probability of an undetected error in the case of a checksum is approximately half of that of using parity with one parity bit per column (i.e. the same number of redundant bits).

While checksums are more effective than parity checks, and are certainly easier to work out in software (since almost all processors have fast adder functions), they can still let an alarming number of undetected errors go through. They're not good enough by themselves, but they are good enough to give a bit of additional protection at the network and higher layers.

1.3.2 The Internet Checksum

The standard form of checksum on the Internet is a slight modification of the basic idea described above. The standard Internet checksum is usually described as the "16-bit one's complement of the one's complement sum" of the relevant parts of the packet. This probably needs a bit of explanation.

The *one's complement* of a number is the result of XORing the number with a number that contains all '1's, in this case (a 16-bit number) that means 0xFFFF in hexadecimal.

The *one's complement sum* of a series of numbers can be obtained by adding up all the numbers, then taking any carried over digits, rotating them right 16-times (so that the first carry bit (bit 16 of the sum) becomes bit zero), and adding them back in to the checksum, and then repeating this process until there are no more carry digits left over.

Usually, the field that the checksum covers contains the checksum itself. In this case, it is important to set the checksum field to zero before calculating the checksum.

An example: consider the short packet we considered above. Calculating the Internet checksum would give:

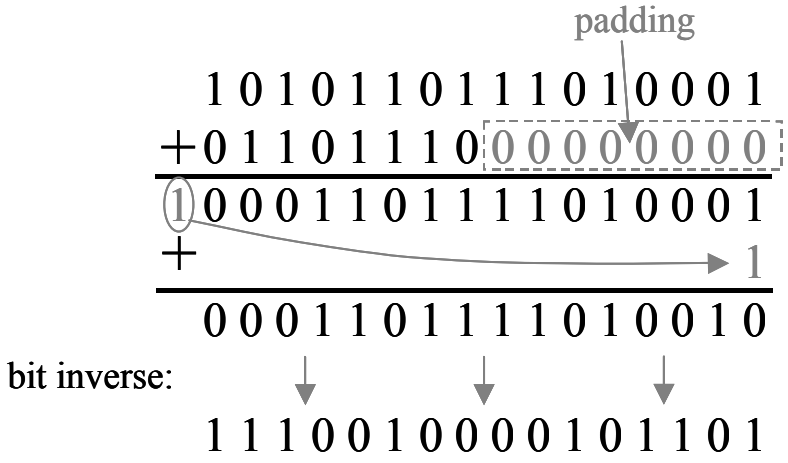

Figure 4 - Calculating the Internet Checksum

Note that with an odd number of bytes in the packet, the packet must be padded with zeros to calculate the checksum.

For anyone who knows C, the following code calculates the Internet checksums for a buffer of length `length`, starting at memory location `buffy`[4]. (Note that this code only works for buffers that are an even number of bytes long: for code that will work for all lengths of buffer, the easiest thing is often to add a zero to the end of the packet, and extend the length by one byte before calculating the checksum.)

```
uint16_t calc_checksum(uint8_t *buffy, uint32_t length) {
uint32_t loop;        // Loop variable to count through buffy
uint32_t sum = 0;     // Accumulator for the checksum

  // Calculate the checksum for this packet:
  for (loop = 0; loop < length; loop += 2) {
    sum += (buffy[loop]<<8) + buffy[loop+1];
  }

  // Then add in carry bits until there is no carry:
  while (sum >> 16) sum = (sum & 0xFFFF) + (sum >> 16);

  // Return the one's complement of the result:
  return (uint16_t) ~sum;
}
```

[4] As with all the code fragments in this text, `uint8_t` is a typedef for an unsigned 8-bit integer, `uint16_t` is an unsigned 16-bit integer, and `uint32_t` is an unsigned 32-bit integer. Anyone using the compiler with the C standard library should have these typedefs defined in the inttypes.h header file.

There are two advantages for this type of checksum: the use of the carry bits in the checksum (rather than just throwing them away) provides slightly greater error protection; and taking the one's complement of the result gives the useful result that performing the checksum calculation on a buffer with a correct checksum in place gives a result of zero, which by convention means "OK, it's worked" in C.

For example, add the calculated checksum above to the start of the packet, and calculate the checksum over the larger five-byte packet that results, and we get:

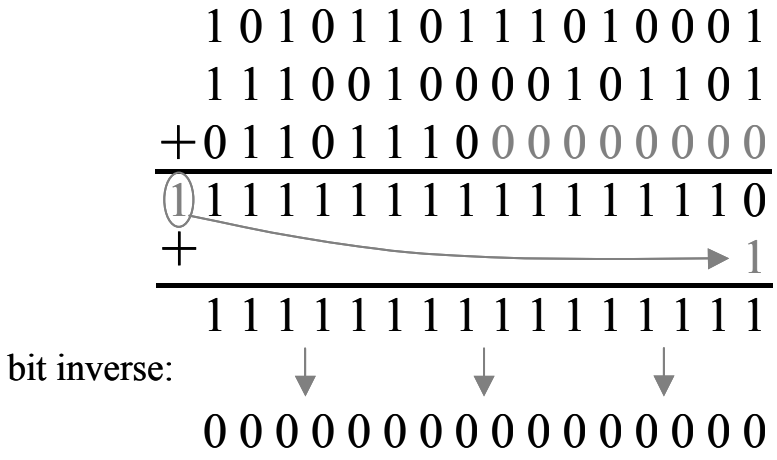

Figure 5 - Checking the Internet Checksum

1.4 The CRC

While checksums are easy to generate in software, and provide some level of protection against bursts of errors, the protection offered is not good enough for most purposes where errors can occur randomly. The technique almost universally used in LANs and most modern communications schemes at the data-link layer is a *cyclic redundancy check* or *CRC*[5]. It has two very important advantages over checksums for use at the data-link layer: it's very easy to compute in hardware, and it provides a much greater level of protection. The maths, however, gets a bit harder.

The idea is that a sequence of bits (called the *Frame Check Sequence* or *FCS*) is placed at the end of a frame, so that the entire frame, including these additional bits, treated as one very long binary number, is exactly divisible by a certain carefully chosen number called the *generator pattern* or *generator polynomial*. The receiver then does a long division of the bits as they come in, and if the final remainder is zero, it assumes the frame was received correctly. (Note: modulo-2 arithmetic is used here: this just means any carried/borrowed bits are ignored. So all additions and subtractions are just exclusive OR operations: very easy in hardware.) Note that if you divide anything by a binary number *(n+1)* bits long, you get a remainder *n* bits long.

[5] Sometimes also confusingly referred to as a "checksum", even though nothing is being added up.

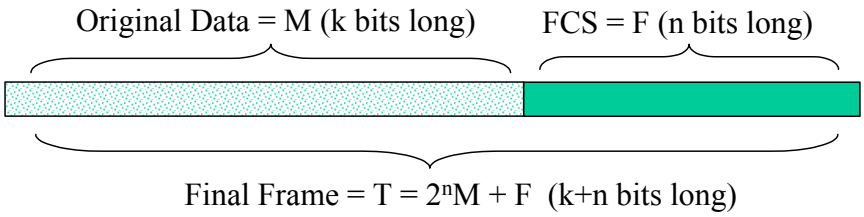

Figure 6 - Adding a FCS to a Frame

That leaves the problem: if you've got a frame, how do you work out what bits to add to it, so that the entire frame (now including the FCS) is divisible by a given generator pattern?

1.4.1 Calculating the FCS

Suppose we have a frame M of length k bits to be transmitted. We want to calculate a CRC of length n bits, such that the total frame T (consisting of M followed by a FCS F) has length $(k + n)$, in other words:

$$T = 2^n M + F \tag{0.9}$$

such that when divided by a pattern P of length $(n+1)$ this number T has no remainder.

So how do we calculate F? In fact it's quite easy: take the original message M, add the right number of zeros (in this case n of them), and then divide by P, and use the remainder as the value of F to be transmitted. In maths:

$$\frac{2^n M}{P} + 0 = Q + \frac{F}{P} \tag{0.10}$$

where Q is some integer. To see that this does give a value for T which is an exact multiple of P:

$$\frac{T}{P} = \frac{2^n M}{P} + \frac{F}{P} = Q + \frac{F}{P} + \frac{F}{P} = Q \tag{0.11}$$

(since any binary number F added to itself modulo-2 gives zero: it's just an exclusive-OR).

1.4.1.1 Example CRC

Consider a simple example: An eight-bit message M = 10101101 and a pattern P = 110101.

First, form 2^5M by adding five zeros to the end of the message, then divide this by P:

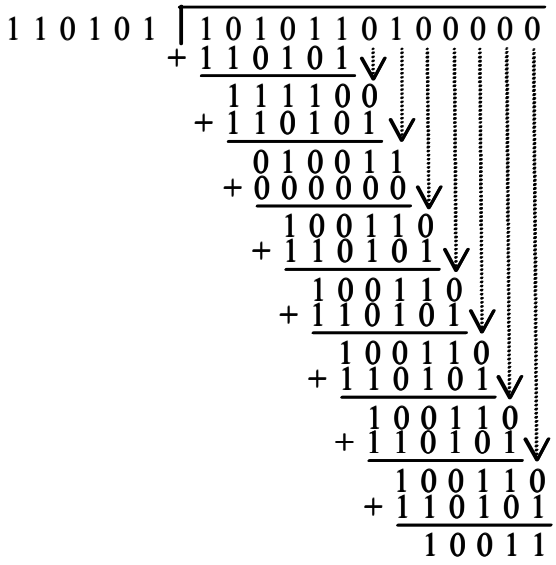

Figure 7 - Example of Computing the FCS

the remainder is 10011, so this is the 5-bit FCS that should be added to the end of the frame *M* before transmission. That gives a total of 13 bits to be transmitted: 1010 1101 10011.

If there are no bit errors, then the received data will be 1010 1101 10011, which is exactly divisible by 110101. Good exercise: check this.

1.4.2 Undetected Errors

Consider a frame protected by a CRC that receives some errors during transmission. Suppose that the pattern of errors are represented by a binary number *E*, which has the same length as the frame, and contains a '1' where an error exists, and a '0' where the bit in the original frame is received correctly. Then (since addition in this one's complement arithmetic is an exclusive-OR operation) the received frame can be represented by:

$$T_r = T + E \tag{0.12}$$

By design, *T* is exactly divisible by the CRC pattern *P*, so for T_r to also be exactly divisible by *P*, *E* must be as well. In other words, for a frame protected by a CRC to be received with an undetected error, the pattern of errors must be divisible by *P*. For a good choice[6] of *P* it can be shown that all errors of the following types are detectable: all single-bit errors, all double bit errors (provided the frame does not exceed a certain length, which for Ethernet is longer than the maximum length frame), any odd number of errors, and any burst of errors up to the length of *P*, and most other errors.

Since most frames require at least four errors to have any chance of the errors being undetected, for any frame of length *k*, protected by a CRC of length *n*, and with a bit error probability so low that the chances of five or more errors can be neglected in comparison with

[6] One for which the CRC pattern has more than one non-zero term, is exactly divisible by 11, and has one prime factor with at least three '1's in it. See Spragins' "Telecommunications Protocols and Design" for further details if interested.

the probability of having four errors, the probability of undetected errors can be approximated by:

$$p_{ue} = \frac{1}{2^n}\binom{k+n}{4} p^4 (1-p)^{k+n-4} \approx \frac{1}{2^n}\frac{(k+n)^4}{24} p^4 (1-p)^{k+n-4} \tag{0.13}$$

So, a 720-bit long frame with a 16-bit CRC (instead of a 16-bit checksum) and a bit error rate of 0.1% would have a rate of undetected errors of approximately 9 x 10^{-8}. In other words, if frames were sent at the rate of a thousand every second, one frame would be received with an undetected error about every four months. That's a bit more like it.

1.4.3 The Problems with Leading and Trailing Zeros

The simple method of calculating the CRC described above has a major problem: it can't distinguish between two frames that are identical apart from the fact that one of them has a few zeros added to the beginning. This doesn't change the long binary number represented by the entire frame, so it doesn't change the remainder when divided by the CRC pattern. If an error results in the addition of any number of zeros to the beginning of a frame protected by a CRC using the algorithm described above, the receiver would not be able to detect the error.

Solutions: add some ones to the beginning of the frame, then calculate the CRC, then throw away the ones. Alternatively, invert the first few bits in the frame before calculating the CRC, so that, in effect, the transmitter is adding a pattern of errors to the frame consisting of *n* '1' bits at the start. Usually, the number of ones added or bits inverted is equal to the length of the FCS field in the frame.

There's another problem as well. Suppose there are some zeros added to the end of the frame after the FCS sequence. This multiplies the long binary number representing the entire frame by a power of two, but it will remain divisible by the CRC pattern. So errors caused by the received frame having a few zeros wrongly added to the end won't be detected either.

Solution: don't transmit the FCS at the end of a frame, transmit the bit-inverse of the FCS instead. In effect, the transmitter is adding a pattern of errors to the frame consisting of *n* '1' bits at the end.

In both cases, the frame can still be checked easily, since the 'correct' result after dividing the entire frame by the generator pattern is now the result from dividing the introduced pattern of errors *E* by the pattern. Add any trailing zeros, however, and the remainder will change.

Ethernet uses both of these techniques to increase the effectiveness of the CRC it uses. This is especially important in the DIX variant of Ethernet, where there is no length field in the frame to tell the receiver how long the frame should be. The length field in the 802.3 CSMA/CD version of Ethernet greatly reduces the probability of any errors due to additional zeros being added to the start or end of frames.

1.4.4 Hardware Implementation

The reason why CRCs are widely used in the lower layers of protocols (notably the data-link layer), but not usually in the higher layers[7], is that they are easy to work out in hardware, but not so easy in software. In hardware, CRC's can be done using a series of XOR gates and a shift register; e.g. consider a CRC pattern of 110101.

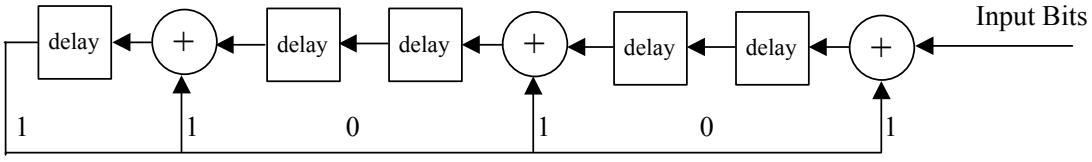

Figure 8 - Hardware CRC Generation

To check that a frame has been received correctly, you just need to input all the bits in the frame in series, and at the end of the frame, all the delay elements in the shift register should contain a 'zero' bit at their output. Clever, isn't it? For an animated demo of this working for simple 3-bit CRCs, try 'Dave's CRC Demo' available from the demos page.

(Note that the Ethernet CRC inverts the first 32 bits of a frame before calculating the CRC, and transmits the bit-inverted FCS, rather than the FCS, so this elegant scheme doesn't quite work. An Ethernet receiver has to calculate the CRC just as the transmitter does, and then compare with the CRC received in the frame. The other oddity about the Ethernet FCS is that it is transmitted most-significant bit first: all other fields in the Ethernet frame are transmitted least-significant bit first.)

1.4.5 Common CRC Generating Patterns

For the longer CRC generating patterns, people usually don't write them out in full as binary numbers. One of two other formats is common: either write them as hexadecimal numbers without the first '1', or in terms of a polynomial, with the powers of x representing the places where there is a '1' in the binary representation. For example, a CRC with P = 101011 could be written in the form 0x0B or $x^5 + x^3 + x + 1$, and P = 10001001 could be written 0x09 or $x^7 + x^3 + 1$. (Note one weakness of the hexadecimal notation: due to the first '1' being missed out, it's impossible to tell from this representation how long the CRC pattern is.)

Project 802 LANs (and Ethernet) use a CRC with:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

which gives P = 1 0000 0100 1100 0001 0001 1101 1011 0111, and generates a 32-bit long FCS field at the end of the frames.

The Universal Serial Bus (USB) uses a CRC with P = 1 1000 0000 0000 0101, giving a 16-bit CRC. Many other CRC patterns of different lengths are also in common use. The demo program 'Dave's Error Detection Demo' uses an 8-bit CRC with P = 1 0000 0111, which is also used by ATM.

---

[7] The Stream Control Transfer Protocol (SCTP) is an exception: this transport layer protocol uses a CRC (see RFC 3309).

1.5 Key Points

- In order of increasing effectiveness, the three most common ways to detect errors are parity, checksums and CRCs.
- Parity and CRC patterns are easy to generate in hardware, but harder in software. Checksums are easy to generate in software, but more difficult in hardware.
- A single parity bit cannot detect any even number of errors, and is particularly bad with burst errors. Checksums are much better at detecting bursts of errors, and have half the undetected error rate of parity checks when errors are random and uncorrelated.
- The Internet checksum is a particularly elegant algorithm that allows the same routine to calculate and check checksums.
- The CRC is a very efficient error-detection system, and easy to implement in hardware.

1.6 Tutorial Questions

1) Prove that changing an odd number of bits in a frame always result in a change in the parity, irrespective of which bits are changed.

2) Give one example each of a communication link which uses parity, checksums and CRC to detect errors. In each case, explain why the choice of that error detection scheme was made.

**3) Suppose that a link using an odd parity check transmitted 4-bit words, and 10% of all frames received failed this parity check. What proportion of frames is being received with undetected errors?

4) Give an example of a pattern of errors in a short packet protected by a sixteen-bit checksum, which would be undetected.

5) Given an original message of "10011011" to be protected by a CRC-pattern of "1011", how many bits are actually transmitted, and what are they?

*6) A message of length 1514 bytes (not including the CRC) is transmitted, protected by a 32-bit CRC (this is the case for maximum length Ethernet frames). The bit error rate on Ethernet is designed to be 10^{-6}. What is the approximate probability that an Ethernet frame is received with an undetected error?

If an Ethernet continually transmitted these frames at the rate of 1,000 per second, what would be the average time between frames being received with undetected errors?

7) A signal protected by a CRC with a pattern of "11001" arrives as "101101001001". Did any errors occur during transmission?

**8) Draw the diagram of a circuit using D-type flip-flops and XOR gates which can determine whether the signal arriving in the last question had any errors in it or not. Explain how the circuit operates.

*9) Is it possible to design an error-detection scheme that can detect <u>all</u> errored frames?

*10) Is it possible to have three errors in a packet, protected by an eight-bit checksum, and not be able to detect the errors? If so, give an example.

**11) You might note in the example code that the data in the packet is added in to the checksum using the line:

```
sum += (buffy[loop]<<8) + buffy[loop+1];
```

where `buffy` is declared as a pointer to an 8-bit integer. Why can't you just declare `buffy` as a pointer to a 16-bit integer, and use the much simpler:

```
sum += buffy[loop];
```

What would the output be if you did use this? Based on this idea, is there a faster way to compute the checksum correctly? (See RFC 1071 for some clues.)

**12) What's the difference between even parity and a one-bit CRC generated using $P = 11$?