

Appendix D

Cyclic Redundancy Check (CRC)

The CRC or *cyclic redundancy check* code is a powerful method of detecting errors in data communications. It is based on an interesting mathematical basis, but is actually quite easy to understand and implement.

A simple example of error detection

Here is a simple example: Suppose you own a mail order business, where you stock several dozen different items, which customers order (by phone) by giving their two-digit stock numbers. For example, item 47 is a hair curler, item 58 is a toothbrush holder, and so on. Every now and then, however, a customer returns an item, saying he got the wrong thing — for example, you shipped item 47 but the customer says he ordered item 49. Somewhere along the way, a 7 in the stock number got changed into a 9.

Your data processing manager now suggests that you change the stock numbers printed in your catalog by adding a third digit at the end; he suggests that this third digit be simply the last digit of the sum of the first two. For example, the sum of the two digits in item 47 (4 plus 7) is 11, the last digit of that sum is 1, and so the new stock number is 471. Item 58 now becomes 583 (5 plus 8 is 13, so put a 3 at the end), item 49 becomes 493, and so on.

What happens the next time a customer orders item 471, but the 7 somehow is changed to a 9 and the number written down is 491 instead? Anyone in your company can spot the fact that there is something wrong because the last digit does not match the first two. Instead of shipping the wrong item, you can now contact the customer and ask him to resubmit his order.

Thus the first two digits of the new stock number (the 47 in 470) is the actual data to be transmitted, and the last digit (the 1) is a check digit, which allows you to detect errors in the data.

Just adding the first two digits to get the check digit is not a good idea, though. It makes it easy for anyone in your company to calculate the check digit, but it also fails to spot a very common human error — the transposition of two digits. Changing 471, for example, to 741 is common, yet 741 is a valid stock number too. To avoid this problem, you might want to change the check digit calculation from a simple sum to perhaps "double the first digit and add it to the second digit" or something like that. With this rule, 47 would become 475 (2 times 4 is 8, plus 7 makes 15, so put a 5 at the end), whereas 74 would become 748. A simple interchange of two digits would no longer generate a valid number.

The problem is that our new formula no longer detects a single digit error. For example, 475 is valid, but so is 975. It points out the need to carefully choose a formula that catches the most errors.

Let's disregard the details, and look at the overall picture. Suppose you implement the check digit scheme with the best possible check digit formula, and then some random error occurs in the number. How likely is it that this scheme will detect the error? For example, let's say an order for item 475 got changed to some random number between 000 and 999 — how likely is it that the new number passes the check digit test and looks like a valid number?

The answer is 10%. That is, a randomly picked check digit can be one of the ten digits from 0 to 9. But once the first two digits of the stock number are picked, there is only one possible check digit that correctly matches the first two. Hence there is only one chance out of 10 that a randomly picked check digit will be the correct one to match the first part of the number. Thus there is a 1 out of 10, or 10% chance of an error being undetected. Looking at it the other way, there is a 9 out of 10 chance, or 90%, that a random error *will* be detected. So our check digit method will spot 90% of all possible errors.

We can then summarize this as follows:

- Our method will spot 90% of all errors.
- It will spot 100% of the simple errors, such as the change of one digit.
- For certain other kinds of errors, it may spot more than 90% of errors, and perhaps as many as 100%.

Similarity to the CRC

The Cyclic Redundancy Check or CRC is the binary equivalent of the check digit scheme discussed above, but with the following differences:

- The data being checked, rather than being just a two-digit stock number, is a (possibly large) number of binary bits.
- Instead of a single check digit, the CRC number (also often called a *Frame Check Sequence* or *FCS*) consists of a number of binary bits, commonly 8, 16, or 32 bits. The calculation is therefore done in binary.
- The formula for calculating the CRC from the given data is carefully chosen to catch as many common errors as possible — the kinds of errors that are most likely to occur in a communications network, such as single bit errors, and strings of consecutive wrong bits.
- The formula is also designed to be easy and quick to calculate using simple hardware. In

modern communications systems, an extremely fast method of computing the CRC bits must be used because systems are usually designed to operate near current "state of the art" speeds; computing the CRC check bits by software in a computer is not fast enough, so a full hardware implementation is needed.

How well the CRC works at catching errors depends on its length. For example, look at an 8-bit CRC code. An 8-bit number can take on one of 256 values, of which only one value is the correct one for any particular set of data. If some random error occurs, the chance is only 1 out of 256 that the CRC pattern will happen to match the given data; hence the likelihood of its *not* matching, and the system therefore detecting the error, is the *other* 255 possible numbers out of the 256, which gives

$$\frac{2^8 - 1}{2^8} = \frac{255}{256} = 99.6\%$$

With a 16-bit CRC, the likelihood of catching errors becomes

$$\frac{2^{16} - 1}{2^{16}} = \frac{65,535}{65,536} = 99.998\%$$

and with a 32-bit CRC, the likelihood becomes

$$\frac{2^{32} - 1}{2^{32}} = \frac{4,294,967,295}{4,294,967,296} = 99.9999998\%$$

Hence a longer CRC greatly improves the chances of catching an error. Just to put that into perspective, imagine that a system makes a random error every $\frac{1}{1000}$ second (which would really be pretty bad in a modern digital system.) That would make 1000 errors per second. An 8-bit CRC would tend to miss one error out of every 256, so it would miss about 4 errors in each second. A 16-bit CRC would tend to miss one error out of every 65,536 so it would take about 656 seconds, or about 11 minutes, between misses. A 32-bit CRC, on the other hand, misses about one error out every 4,294,967,296; it would be 4,294,967 seconds or almost 50 days between misses.

These numbers are actually quite conservative, because they assume that all these errors are truly random and major. Today's digital communications systems make much fewer than 1000 errors per second, and many of these errors are the type that are guaranteed to be caught (such as single-bit errors or errors involving a small number of consecutive bits — what is called a *burst error*), so even a 16-bit CRC is usually considered to be good enough.

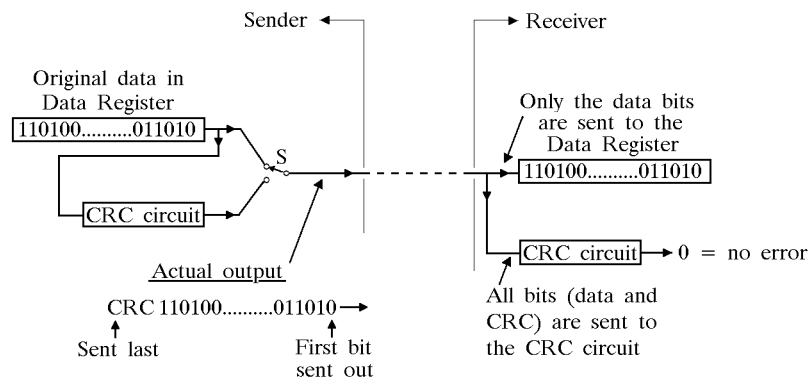


Fig. D-1. CRC is added and checked like this

How is CRC implemented?

The CRC is used a bit differently from what some people (including authors of some other textbooks!) think. Look at Fig. D-1 as we explain.

The sender (at the left) has some binary data to send to the receiver at the right. The original data is placed into a data register. This is a group of flip-flops, connected as a shift register.

DETOUR

We explain registers and flip-flops elsewhere in the book, but here is a short explanation anyway in case you need it.

A *flip-flop* is a fairly simple circuit which can hold one bit of data. It usually consists of some transistors and resistors, and is often part of an integrated circuit. It has some inputs and also some outputs. If you put a one or zero into it, it will remember that bit as long as the power is on, or as long as you don't overwrite it with another bit.

A group of flip-flops that holds a group of bits is called a *register*; if it holds data, then it is called a *data register*. You need one flip-flop for each bit to be stored.

A group of flip-flops can also be connected in a row as a *shift register*. In this case, the bits stored in the flip-flops can be made to "hop" from one flip-flop to the next in response to pulses from a clock oscillator. This is very similar to a bunch of kids playing musical chairs — at the beat of drum, everyone gets up from the current chair and moves into the next chair over — except that in musical chairs the chairs are arranged in a circle, whereas the flip-flops are arranged in a row and all the bits move in the same direction. As each bit moves over,

a new place opens at one end of the register for possibly a new bit, whereas the bit at the opposite end “falls off” the other end of the register, and becomes an output bit.



Both of the data registers in Fig. D-1 are wired as shift registers. The one in the sender outputs its bits, one at a time, while the one in the receiver inputs the bits as they come in. Notice the order in which they come out — the bits at the right end of the data register come out first, so the order (for the example in Fig. D-1) is 0, then 1, then 0, then two 1’s, and so on.

Given enough clock pulses, all the bits from the sender’s shift register will eventually be transferred into the receiver’s shift register (through switch S, which is initially up.)

While all this is happening, the bits coming out the sender’s shift register are also going into its CRC circuit. The CRC circuit (which started out empty) gradually starts receiving data bits and computing its CRC at the same time. When all the bits from the data register have been sent out, switch S flips into the down position, and the CRC bits (8, 16, 32, or whatever length CRC is being used) are then sent out as the end of the outgoing bit stream.

Thus the output from the sender is all of the data bits (with the rightmost bit coming out of the data register first), and then followed by the CRC bits. These bits travel to the receiver one-by-one, through some kind of a connection, shown as a dotted line in Fig. D-1. They arrive at the receiver, hopefully without error.

As they arrive at the receiver, the first group (consisting of all the data bits) is shifted into its data register. At the same time, the same bits are sent into the receiver’s CRC circuit. This CRC circuit is identical to the one at the sender (we will discuss its wiring in a moment).

Just like the sender’s CRC circuit, the receiver’s also starts out empty. If all the incoming bits are exactly the same as what the sender sent, then at the end of all the data bits, the receiver’s CRC circuit should be holding exactly the same CRC as the sender’s did. Remember — it’s exactly the same circuit, it starts out in exactly the same empty state, and it receives (hopefully) the same incoming bits. So it should end up containing exactly the same bits. It would therefore be possible to compare its output with the incoming CRC bits, and assume that there was no error if they are, in fact, exactly equal.

Many textbooks describe the CRC process exactly this way, but in real life things often happen just a bit differently. Read on...

Not shown in Fig. D-1 is an extra signal path that sends some additional ZEROS to the CRC circuit in the sender. The number of ZEROS is equal to the size of the CRC code. For example, if there are 16 bits in the data and the CRC generates five bits, then the sender’s CRC circuit gets a total of 21 bits — the 16 data bits and then 5 additional ZEROS. The CRC number is what remains in the CRC circuit after those five ZEROS have been entered.

As Fig. D-1 shows, the receiver’s CRC circuit gets not only the data bits, but also the sender’s CRC bits; in our example, therefore, the receiver’s CRC circuit would also get 21 bits — 16 data bits plus 5 CRC bits. The CRC circuit is so designed that if the incoming bits are the correct CRC for the previous data, then the CRC circuit resets itself to all ZEROS.

Therefore, if, after all this is over, the output from the receiver’s CRC circuit is all zero, this means that the incoming CRC code (computed from the original data) matches the receiver’s CRC (computed from the received data), and so it is assumed that the received data was identical to the original data bits, and there was no error.



There is another way to look at this. Let’s again assume 16 data bits plus a 5-bit CRC, although this discussion would work for any numbers of bits.

If there is no error in the data, then after the first 16 bits are received (the data bits), both the sender’s CRC circuit and the receiver’s CRC circuit will have the same 5-bit number in it. Using this as a starting point, if you then send the code 00000 into the CRC circuit, it will change to some new number; let’s call it GHJKL, where each of the letters is a bit. If, on the other hand, you start with the same CRC number, but send in the GHJKL bits, it will do the exact opposite and reset itself back to 00000.



The CRC Circuit

So what is in the two CRC boxes in Fig. D-1?

In order for the CRC circuit to be able to keep up with the rest of the system (which is probably pushing its hardware to the limits to begin with), the CRC circuit has to be fast and simple.

The circuit is a slightly modified shift register. It has a number of flip-flops, shown as rectangles in Fig. D-2. The number of flip-flops is the same as the CRC length, so an 8-bit CRC would require 8 flip-flops, and so on.

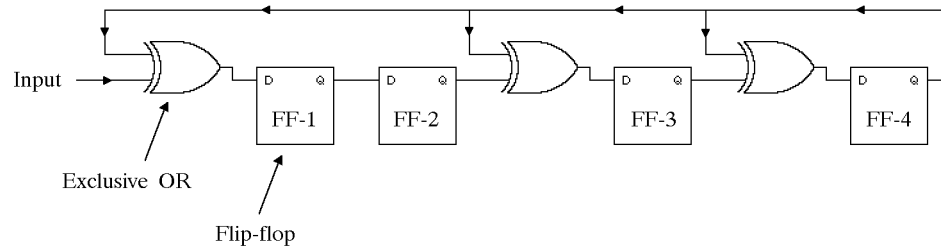


Fig. D-2. Circuit to generate CRC code

Incoming data comes in from the left side of the circuit, and gradually shifts from flip-flop to flip-flop, moving to the right. But the number of incoming bits is generally much larger than the length of the CRC (and the number of flip-flops), so there is no room to store all incoming bits. Instead, once the shift register fills up with a few bits, each incoming bit coming in from the left pushes a bit out from the right end. With a plain shift register, these bits would simply be shifted out and lost.

In the CRC circuit, however, these bits are not lost, because bits being pushed out from the right end are brought back and mixed in with the incoming bits in various places, via a group of exclusive-OR gates.

DETOUR →

An exclusive-OR gate is a digital circuit with two inputs and one output. If the two inputs are the same — both 0 or both 1 — then the gate outputs a 0. If the two inputs are different — one of them is a 0 and the other a 1 — then the gate outputs a 1.

← **END OF DETOUR**

As a result of the mixing process, new data is continually being mixed with old data in various ways. After all the incoming bits have come in, the bits remaining in the flip-flops are a peculiar mixture of all the bits that have come in since the start. This is the CRC code.

The whole thing probably seems very mysterious. The important thing to remember is that the sender's CRC circuit and the receiver's CRC circuit are identical, and so they produce the same results. Actually, the circuit has been carefully designed, and the exclusive-OR gates are carefully placed so as to provide the proper CRC code.

Equipment designers use a shorthand notation to indicate how many flip-flops there are, and where the exclusive-ORs should be connected. The notation for the simple circuit of Fig. D-2 (which is not a real working circuit — there are a few missing connections and, with just four flip-flops, it's also too small for general use) would be

$$X^4 + X^3 + X^2 + 1$$

- In this notation,
- X^4 says there are four flip-flops, (and the output of the last one goes back to all the exclusive-OR gates)
 - X^3 and X^2 says there are exclusive-OR gates at the outputs of flip-flops 3 and 2, and
 - the 1 says there is also an exclusive-OR at the very input

Since sending and receiving equipment is often made by different manufacturers, it's obviously essential that all these companies get together and agree on using the same CRC circuit. Several specific CRC circuits have therefore come into standard use. In the United States, there are

CRC-8:

$$X^8 + X^2 + X^1 + 1$$

CRC-16:

$$X^{16} + X^{15} + X^2 + 1$$

and CRC-32, which is:

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

A common European circuit is CRC-CCITT:

$$X^{16} + X^{12} + X^5 + 1.$$

Given these equations, you can easily see how the circuit itself would have to be connected.

A Spreadsheet simulation of a CRC circuit

A spreadsheet program, such as Excel or 123, can easily be programmed to simulate the operation of a CRC circuit. If you have a computer with a spreadsheet program, follow the steps below to see how the circuit of Fig. D-2 operates to implement the equation

$$X^4 + X^3 + X^2 + 1$$

There are thus four flip-flops, and let's assume that the data is 8 bits long. We will use five columns and 14 rows, as follows:

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					

Now we set up our column headings into cells A1 through E1:

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2					

Column A will hold the input bits, while columns B through E are used for the four flip-flops. Next, put in the initial state for the four flip-flops, which is 0000:

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2		0	0	0	0
3					

Next, let's make up some eight-bit data to be sent, such as perhaps 01110010. Since there are four flip-flops, we will add four ZEROS at the end, so there are 12 bits to be entered into the Input column, like this:

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2	0	0	0	0	0
3	1				
4	1				
5	1				
6	0				
7	0				
8	1				
9	0				
10	0				
11	0				
12	0				
13	0				
14					

Now it is time to specify the wiring of the circuit. The CRC circuit is run by a clock signal which "ticks" once for each data bit. On each "tick", all the flip-flops look at their D inputs and decide whether to change to a ZERO or a ONE state.

To start off, let's assume that row 2 of the spreadsheet is "now" or the present state, whereas row 3 just below it will be the "next" state. Looking at Fig. D-2, we see that the output of FF-1 goes directly to the input of FF-2, so whatever is in FF-1 now will go into FF-2 next. Thus whatever is in cell B2 should go into C3 on the next state. So into cell C3 put the equation +B2. If you do this properly, cell C3 will change to a 0, like this:

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2	0	0	0	0	0
3	1		0		
4	1				

Looking back at Fig. D-2, we see that the input to FF-1 (which will determine the next state of FF-1 in cell B3) is the exclusive-OR of the present input (in cell A2) and the present output from FF4 (which is in cell E2).

The logical Exclusive-OR function is defined as 0 if the two inputs are the same, and 1 if they are different. Using a math formula, this can best be described as

$$|A2 - E2|$$

If the contents of cell A2 is the same as the contents of E2, then the result is 0; if one of them is 0 and the other is 1 (which are the only possible values in a binary system), then the result is 1. Using spreadsheet symbols, put the formula @ABS(A2 - E2) into cell B3; this will

put a 0 into that cell since A2 and E2 both contain 0 and are thus the same.

In the same way, the next state of FF-3 in cell D3 is the exclusive-OR of FF-2 and FF-4, so put in the equation @ABS(C2 – E2). Likewise, into cell E3 put the equation @ABS(D2 – E2). You will now get the following data, which indicates that after the first clock “tick”, the four flip-flops will still be 0000:

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2	0	0	0	0	0
3	1	0	0	0	0
4	1				

So far, we see that row 2 represents the input data and the four flip-flops at the very beginning (before the first clock pulse), whereas row 3 represents the state *after* that first clock pulse.

In exactly the same way, row 4 should represent the data and flip-flops after the second clock pulse, so simply copy cells B3 through E3 into B4 through E4. Then copy B4 through E4 into B5 through E5, and so on, all the way down through row 14. Actually, you can do all these copies in one step, rather than do just one row at a time. This will then give the following

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2	0	0	0	0	0
3	1	0	0	0	0
4	1	1	0	0	0
5	1	1	1	0	0
6	0	1	1	1	0
7	0	0	1	1	1
8	1	1	0	0	0
9	0	1	1	0	0
10	0	0	1	1	0
11	0	0	0	1	1
12	0	1	0	1	0
13	0	0	1	0	1
14		1	0	0	1

Row 14 at the bottom shows the final state of the four flip-flops; the four CRC bits are therefore 1001.

Let’s just quickly review before we go on. Column A shows that the input into the sender’s CRC circuit is the eight data bits 01110010, followed by an extra 0000 sequence in rows 10 through 13. Columns B through E show the successive states of the four flip-flops, and row 14 shows the final CRC output of 1001.

Note one thing: although the sender’s CRC generator got the eight data bits followed by 0000, the actual

data sent to the receiver has the CRC bits 1001, not the 0000 at the very end. Hence the actual data sent is 01110010 and then 1001.

Let’s now look at the receiver’s CRC circuit. It is wired up exactly the same way, so all of the equations entered into the spreadsheet should still apply. But it gets a different set of input bits. If no error occurred, then it gets 01110010 and then 1001. So let’s go back to the spreadsheet, but replace the 0000 in A10 through A13 with 1001, like this:

	A	B	C	D	E
1	Input	FF-1	FF-2	FF-3	FF-4
2	0	0	0	0	0
3	1	0	0	0	0
4	1	1	0	0	0
5	1	1	1	0	0
6	0	1	1	1	0
7	0	0	1	1	1
8	1	1	0	0	0
9	0	1	1	0	0
10	1	0	1	1	0
11	0	1	0	1	1
12	0	1	1	1	0
13	1	0	1	1	1
14		0	0	0	0

As expected, we get 0000 at the end.

We therefore see that, if we enter the data bits plus 0000 in the sender, we get the CRC. If we enter the same data bits plus the CRC in the receiver, then we get 0000.

You can now experiment with the spreadsheet data. Change any one or more bits in the Input column, and the last column is no longer 0000. In other words, anything other than 0000 at the end indicates an error.

But does the CRC catch *all possible* errors? No — you can see that for yourself. Place 12 ZEROS into the Input column, and you will see that the last row is also 0000. Or try 111011101001 or 111110011111; those also give 0000. In fact, there are 256 possible input combinations which all give 0000 in the bottom row.

We can come to that conclusion as follows: With a 4-bit CRC, we have 2^4 or 16 possible CRC combinations for any particular set of data, of which only one is correct. Hence given a random error, there is a 1 out of 16 chance that the CRC will appear correct even though there is an error in transmission.

Now, although there are only eight data bits, there are actually 12 bits being sent counting the four-bit CRC. Hence there are 2^{12} or 4096 possible combinations of data plus CRC that could be received, and one out of every 16 of those will appear to have the correct

CRC even though it is wrong. One sixteenth of 4096 is 256.

This points out that a four-bit CRC will let through one out of every 16 errors — not good enough for most systems. Hence most CRC systems use more bits. You could simulate a longer CRC on a spreadsheet without too much difficulty, but even a four-bit example is good enough to prove the point.