Cyclic Redundancy Checks

One of the most popular methods of error detection for digital signals is the Cyclic Redundancy Check (CRC). The basic idea behind CRCs is to treat the message string as a single binary word M, and divide it by a key word k that is known to both the transmitter and the receiver. The remainder r left after dividing M by k constitutes the "check word" for the given message. The transmitter sends both the message string M and the check word r, and the receiver can then check the data by repeating the calculation, dividing M by the key word k, and verifying that the remainder is r. The only novel aspect of the CRC process is that it uses a simplified form of arithmetic, which we'll explain below, in order to perform the division.

By the way, this method of checking for errors is obviously not foolproof, because there are many different message strings that give a remainder of r when divided by k. In fact, about 1 out of every k randomly selected strings will give any specific remainder. Thus, if our message string is garbled in transmission, there is a chance (about 1/k, assuming the corrupted message is random) that the garbled version would agree with the check word. In such a case the error would go undetected. Nevertheless, by making k large enough, the chances of a random error going undetected can be made extremely small.

That's really all there is to it. The rest of this discussion will consist simply of refining this basic idea to optimize its effectiveness, describing the simplified arithmetic that is used to streamline the computations for maximum efficiency when processing binary strings.

When discussing CRCs it's customary to present the key word k in the form of a "generator polynomial" whose coefficients are the binary bits of the number k. For example, suppose we want our CRC to use the key k=37. This number written in binary is 100101, and expressed as a polynomial it is $x^5 + x^2 + 1$. In order to implement a CRC based on this polynomial, the transmitter and receiver must have agreed in advance that this is the key word they intend to use. So, for the sake of discussion, let's say we have agreed to use the generator polynomial 100101.

By the way, it's worth noting that the remainder of any word divided by a 6-bit word will contain no more than 5 bits, so our CRC words based on the polynomial 100101 will always fit into 5 bits. Therefore, a CRC system based on this polynomial would be called a "5-bit CRC". In general, a polynomial with k bits leads to a "k-1 bit CRC".

Now suppose I want to send you a message consisting of the string of bits M = 00101100010101110100011, and I also want to send you some additional information that will allow you to check the received string for correctness. Using our agreed key word k=100101, I'll simply "divide" M by k to form the remainder r, which will constitute the CRC check word. However, I'm going to use a simplified kind of division that is particularly well-suited to the binary form in which digital data is expressed.

If we interpret k as an ordinary integer (37), it's binary representation, 100101, is really shorthand for

$$(1)2^5 + (0)2^4 + (0)2^3 + (1)2^2 + (0)2^1 + (1)2^0$$

Every integer can be expressed uniquely in this way, i.e., as a polynomial in the base 2 with coefficients that are either 0 or 1. This is a very powerful form of representation, but it's actually more powerful than we need for purposes of performing a data check. Also, operations on numbers like this can be somewhat laborious, because they involve borrows and carries in order to ensure that the coefficients are always either 0 or 1. (The same is true for decimal arithmetic, except that all the digits are required to be in the range 0 to 9.)

To make things simpler, let's interpret our message M, key word k, and remainder r, not as actual integers, but as abstract polynomials in a dummy variable x (rather than a definite base like 2 for binary numbers or 10 for decimal numbers). Also, we'll simplify even further by agreeing to pay attention only to the parity of the coefficients, i.e., if a coefficient is an odd number we will simply regard it as 1, and if it is an even number we will regard it as 0. This is a tremendous simplification, because now we don't have to worry about borrows and carries when performing arithmetic. This is because every integer coefficient must obviously be either odd or even, so it's automatically either 0 or 1.

To give just a brief illustration, consider the two polynomials $x^2 + x + 1$ and $x^3 + x + 1$. If we multiply these together by the ordinary rules of algebra we get

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

but according to our simplification we are going to call every 'even' coefficient 0, so the result of the multiplication is simply $x^5 + x^4 + 1$. You might wonder if this simplified way of doing things is really self-consistent. For example, can we divide the product $x^5 + x^4 + 1$ by one of its factors, say, $x^2 + x + 1$, to give the other factor? The answer is yes, and it's much simpler than ordinary long division. To divide the polynomial 110001 by 111 (which is the shorthand way of expressing our polynomials) we simply apply the bit-wise exclusive-OR operation repeatedly as follows

```
           1011
         ______
    111 |110001
         111
         ---
         0010
          000
          ---
          0100
           111
           ----
           0111
            111
```

```
             ---
             000
```

This is exactly like ordinary long division, only simpler, because at each stage we just need to check whether the leading bit of the current three bits is 0 or 1. If it's 0, we place a 0 in the quotient and exclusively OR the current bits with 000. If it's 1, we place a 1 in the quotient and exclusively OR the current bits with the divisor, which in this case is 111. As can be seen, the result of dividing 110001 by 111 is 1011, which was our other factor, x^3 + x + 1, leaving a remainder of 000. (This kind of arithmetic is called the arithmetic of polynomials with coefficients from the field of integers modulo 2.)

So now we're armed with everything we need to actually perform a CRC calculation with the message string M and key word k defined above. We simply need to divide M by k using our simplified polynomial arithmetic. In fact, it's even simpler, because we don't really need to keep track of the quotient - all we really need is the remainder. So we simply need to perform a sequence of 6-bit "exclusive ORs" with our key word k, beginning from the left-most "1 bit" of the message string, and at each stage thereafter bringing down enough bits from the message string to make a 6-bit word with leading 1. A worksheet for the entire computation is shown below:

```
          _____________________
 100101 |00101100010101110100011
           100101
           ------
           00100101
             100101
             ------
             0000000101110
                    100101
                    ------
                    00101110
                      100101
                      ------
                      00101100
                        100101
                        ------
                        00100111
                          100101
                          ------
                          000010   remainder = CRC
```

Our CRC word is simply the remainder, i.e., the result of the last 6-bit exclusive OR operation. Of course, the leading bit of this result is always 0, so we really only need the last five bits. This is why a 6-bit key word leads to a 5-bit CRC. In this case, the CRC word for this message string is 00010, so when I transmit the message word M I will also send this corresponding CRC word. When you receive them you can repeat the above calculation on M with our agreed generator polynomial k and verify that the resulting remainder agrees with the CRC word I included in my transmission.

What we've just done is a perfectly fine CRC calculation, and many actual implementations work exactly that way, but there is one

potential drawback in our method. As you can see, the computation described above totally ignores any number of "0"s ahead of the first "1" bit in the message. It so happens that many data strings in real applications are likely to begin with a long series of "0"s, so it's a little bothersome that the algorithm isn't working very hard in such cases. To avoid this "problem", we can agree in advance that before computing our n-bit CRC we will always begin by exclusive ORing the leading n bits of the message string with a string of n "1"s. With this convention (which of course must be agreed by the transmitter and the receiver in advance) our previous example would be evaluated as follows

```
   00101100010101110100011   <--  Original message string
   11111                      <-- "Fix" the leading bits
   -----------------------
   11010100010101110100011    <-- "Fixed" message string
   100101
   ------
   0100000
    100101
    ------
    000101001
       100101
       ------
       00110001
         100101
         ------
         0101000
          100101
          ------
          00110111
            100101
            ------
            0100101
             100101
             ------
             0000000100011
                    100101
                    ------
                    000110   remainder = CRC
```

So with the "leading zero fix" convention, the 5-bit CRC word for this message string based on the generator polynomial 100101 is 00110. That's really all there is to computing a CRC, and many commercial applications work exactly as we've described. People sometimes use various table-lookup routines to speed up the divisions, but that doesn't alter the basic computation or change the result. In addition, people sometimes agree to various non-standard conventions, such as interpreting the bits in reverse order, but the essential computation is still the same. (Of course, it's crucial for the transmitter and receiver to agree in advance on any unusual conventions they intend to observe.)

Now that we've seen how to compute CRC's for a given key polynomial, it's natural to wonder whether some key polynomials work better (i.e., give more robust "checks") than others. From one point of view the answer is obviously yes, because the larger our key word, the less likely it is that corrupted data will go undetected. By appending an n-bit CRC to our message string we are increasing the

total number of possible strings by a factor of 2^n, but we aren't increasing the degrees of freedom, since each message string has a unique CRC word. Therefore, we have established a situation in which only 1 out of 2^n total strings (message+CRC) is valid. Notice that if we append our CRC word to our message word, the result is a multiple of our generator polynomial. Thus, of all possible combined strings, only multiples of the generator polynomial are valid.

So, if we assume that any corruption of our data affects our string in a completely random way, i.e., such that the corrupted string is totally uncorrelated with the original string, then the probability of a corrupted string going undetected is 1/(2^n). This is the basis on which people say a 16-bit CRC has a probability of 1/(2^16) = 1.5E-5 of failing to detect an error in the data, and a 32-bit CRC has a probability of 1/(2^32), which is about 2.3E-10 (less than one in a billion).

Since most digital systems are designed around blocks of 8-bit words (called "bytes"), it's most common to find key words whose lengths are a multiple of 8 bits. The two most common lengths in practice are 16-bit and 32-bit CRCs (so the corresponding generator polynomials have 17 and 33 bits respectively). A few specific polynomials have come into widespread use. For 16-bit CRCs one of the most popular key words is 10001000000100001, and for 32-bit CRCs one of the most popular is 100000100110000010001110110110111. In the form of explicit polynomials these would be written as

$$x^{16} + x^{12} + x^5 + 1$$

and

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

The 16-bit polynomial is known as the "X25 standard", and the 32-bit polynomial is the "Ethernet standard", and both are widely used in all sorts of applications. (Another common 16-bit key polynomial familiar to many modem operators is 11000000000000101, which is the basis of the "CRC-16" protocol.) These polynomials are certainly not unique in being suitable for CRC calculations, but it's probably a good idea to use one of the established standards, to take advantage of all the experience accumulated over many years of use.

Nevertheless, we may still be curious to know how these particular polynomials were chosen. It so happens that one could use just about ANY polynomial of a certain degree and achieve most of the error detection benefits of the standard polynomials. For example, ANY n-bit CRC will certainly catch any single "burst" of m consecutive "flipped bits" for any m less than n, basically because a smaller polynomial can't be a multiple of a larger polynomial. Also, we can ensure the detection of any odd number of bits simply by using a generator polynomial that is a multiple of the "parity polynomial", which is x+1. A polynomial of our simplified kind is a multiple of x+1 if and only if it has an even number of terms.

It's interesting to note that the standard 16-bit polynomials both

include this parity check, whereas the standard 32-bit CRC does not. It might seem that this represents a shortcoming of the 32-bit standard, but it really doesn't, because the inclusion of a parity check comes at the cost of some other desirable characteristics. In particular, much emphasis has been placed on the detection of two separated single-bit errors, and the standard CRC polynomials were basically chosen to be as robust as possible in detecting such double-errors. Notice that the basic "error word" E representing two erroneous bits separated by j bits is of the form x^j + 1 or, equivalently, x^j - 1. Also, an error E superimposed on the message M will be undetectable if and only if E is a multiple of the key polynomial k. Therefore, if we choose a key that is not a divisor of any polynomial of the form x^t - 1 for t=1,2,...,m, then we are assured of detecting any occurrence of precisely two erroneous bits that occur within m places of each other.

How would we find such a polynomial? For this purpose we can use a "primitive polynomial". For example, suppose we want to ensure detection of two bits within 31 places of each other. Let's factor the error polynomial x^31 - 1 into it's irreducible components (using our simplified arithmetic with coefficients reduced modulo 2). We find that it splits into the factors

```
x^31 - 1  =   (x+1)
            *(x^5 + x^3 + x^2 + x + 1)
            *(x^5 + x^4 + x^2 + x + 1)
            *(x^5 + x^4 + x^3 + x + 1)
            *(x^5 + x^2 + 1)
            *(x^5 + x^4 + x^3 + x^2 + 1)
            *(x^5 + x^3 + 1)
```

Aside from the parity factor (x+1), these are all primitive polynomials, representing primitive roots of x^31 - 1, so they cannot be divisors of any polynomial of the form x^j - 1 for any j less than 31. Notice that x^5 + x^2 + 1 is the generator polynomial 100101 for the 5-bit CRC in our first example.

Another way of looking at this is via recurrence formulas. For example, the polynomial x^5 + x^2 + 1 corresponds to the recurrence relation s[n] = (s[n-3] + s[n-5]) modulo 2. Beginning with the initial values 00001 this recurrence yields

```
                                    |--> cycle repeats
      0000100101100111110001101110101 00001
```

Notice that the sequence repeats with a period of 31, which is another consequence of the fact that x^5 + x^2 + 1 is primitive. You can also see that the sets of five consecutive bits run through all the numbers from 1 to 31 before repeating. In contrast, the polynomial x^5 + x + 1 corresponds to the recurrence s[n] = (s[n-4] + s[n-5]) modulo 2, and gives the sequence

```
                           |--> cycle repeats
      000010001100101011111 00001
```

Notice that this recurrence has a period of 21, which implies that the polynomial x^5 + x + 1 divides x^21 - 1. Actually, x^5 + x + 1 can be factored as (x^2 + x + 1)(x^3 + x^2 + 1), and both of those factors divide x^21 - 1. Therefore, the polynomial x^5 + x + 1 may

be considered to give a less robust CRC than x^5 + x^2 + 1, at least from the standpoint of maximizing the distance by which two erroneous bits must be separated in order to go undetected.

On the other hand, there are error patterns that would be detected by x^5 + x + 1 but would NOT be detected by x^5 + x^2 + 1. As noted previously, any n-bit CRC increases the space of all strings by a factor of 2^n, so a completely arbitrary error pattern really is no less likely to be detected by a "poor" polynomial than by a "good" one. The distinction between good and bad generators is based on the premise that the most likely error patterns in real life are NOT entirely random, but are most likely to consist of a very small number of bits (e.g., one or two) very close together. To protect against this kind of corruption, we want a generator that maximizes the number of bits that must be "flipped" to get from one formally valid string to another. We can certainly cover all 1-bit errors, and with a suitable choice of generators we can effectively cover virtually all 2-bit errors.

Whether this particular failure mode deserves the attention it has received is debatable. If our typical data corruption event flips dozens of bits, then the fact that we can cover all 2-bit errors seems less important. Some cynics have gone so far as to suggest that the focus on the "2-bit failure mode" is really just an excuse to give communications engineers an opportunity to deploy some non-trivial mathematics. I personally wouldn't go quite that far, since I believe it makes sense to use a primitive generator polynomial, just as it would make sense to use a prime number key if we were working with ordinary integer arithmetic, because one big coincidence seems intuitively less likely than several small ones. However, the fact remains that our overall estimate for the probability of an error going undetected by an n-bit CRC is 1/(2^n), regardless of which (n+1)-bit generator polynomial we use.

The best argument for using one of the industry-standard generator polynomials may be the "spread-the-blame" argument. Any CRC (like a pseudo-random number generator) COULD be found to be particularly unsuitable in some special circumstance, e.g., in an environment that tends to produce error patterns in multiples of our generator at a rate significantly greater than would be predicted for a truly random process. This would be incredibly bad luck, but if it ever happenned, you'd like to at least be able to say you were using an industry standard generator, so the problem couldn't be attributed to any unauthorized creativity on your part.

Return to MathPages Main Menu