The Great CRC Mystery

Terry Ritter

The Cyclic Redundancy Check (or CRC), is a way to detect errors in data storage or transmission. With more and more data being transmitted over phone lines, the need for protocols that protect data from damage in transit as increased, but the theory behind CRC generation is not well known.

What Is a CRC?

The Cyclic Redundancy Check is a way to detecting small changes in blocks of data. Error detection is especially important when computer programs are transmitted or stored, because an error of even one bit (perhaps out of hundreds of thousands) is often sufficient to make a program faulty. Although a few errors in a text file might be acceptable (since the text can be reedited when received or recovered), an error-free file is preferable. An error-correcting protocol triggered by CRC error-detection can provide this accuracy at low cost.

The CRC algorithm operates on a block of data as a unit [1]. We can understand the CRC better if we see a block of data as a single (large) numerical value. The CRC algorithm divides this large value by a magic number (the CRC polynomial or generator polynomial), leaving the remainder, which is our CRC result.

The CRC result can be sent or stored along with the original data. When the data is received (or recovered from storage) the CRC algorithm can be reapplied, and the latest result compared to the original result. If an error has occurred, we will probably get a different CRC result. Most uses of CRC do not attempt to classify or locate the error (or errors), but simply arrange to repeat the data operation until no errors are detected.

Using the CRC

The IBM 8-inch floppy disk specification used the CRC-CCITT polynomial for error-detection, and this CRC is now used in almost all disk controller devices. A disk controller computes a CRC as it writes a disk sector, and then it appends that CRC to the data. When the data is read back, a new CRC is computed from the recovered data and compared to the original CRC. If the CRC values differ, an error has occurred and the operation is repeated. The standard disk CRC (CRC-CCITT) is hidden in the controller, and nowadays receives little comment.

One version of the XMODEM (or Christensen) file transmission protocol uses the CRC-CCITT polynomial to detect data transmission errors, typically caused by line noise. When the receiving end detects a data error, it sends a NAK (Negative AcKnowledge) character to the sender, which requests that the data block be retransmitted. The receiving end repeats this process until the CRC from the transmitting end matches the local result, or until one or both ends give up. When the result does match, the receiving end sends an ACK (ACKnowledge) character, and the transmitting end then sends the next block.

Error Control and Efficiency

Many different CRC polynomials are possible; these generator polynomials are designed and constructed to have desirable error-detection properties. If the CRC polynomials are "well constructed" the major difference between them is in their length. Longer polynomials provide more assurance of data accuracy and are fully usable over larger amounts of data; however, longer polynomials also produce longer remainder values, which add additional error-checking overhead to the data.

A "16-bit" polynomial has a 16-bit remainder. There are two well-known 16-bit polynomials: CRC-16 (used in early BISYNC protocols) and CRC-CCITT (used in disk storage, SDLC, and XMODEM CRC). Of the two, CRC-CCITT may be a little stronger, and, by convention is often used in ways which strengthen its error-detection capabilities. This article illustrates CRC-CCITT, which is the polynomial x^16 + x^12 + x^5 + 1.

Polynomials are classified by their highest non-zero digit (or place) which is termed the degree of the polynomial. Both CRC-16 and CRC-CCITT are of degree 16, which means that bits 16 through 0 are significant in their description; a degree 16 polynomial thus has 17 bits. Normally we are most concerned with the remainder of the CRC operation, which has one bit less than the polynomial. Thus, we may think of 16-bit CRC's, even though their generator polynomials actually contain 17 bits (bits 16 through 0).

In a proper CRC polynomial, both the most significant bit (MSb) and least significant bit (LSb) are always a '1'. Because the highest bit of the polynomial is always a '1', we are able to treat this bit differently from the other bits of the polynomial. Since the remainder from a 16th degree polynomial has only 16 bits, a 16-bit register is sufficient for CRC operations on a 16-bit polynomial, even though the polynomial itself actually has 17 bits.

A well-constructed CRC polynomial over limited-size data blocks will detect any contiguous burst of errors shorter than the polynomial, any odd number of errors throughout the block, any 2 bit errors anywhere in the block, and most other cases of any possible errors anywhere in the data [2]. So every possible arrangement of 1, 2, or 3 bit errors will be detected. Nevertheless, there remains a small possibility that some errors will not be detected. This happens when the pattern of the errors results in a new value which, when divided, produces exactly the same remainder as the correct block. With a properly constructed 16-bit CRC, there is an average of one error pattern which will not be detected for every 65,535 which would be detected. That is, with CRC-CCITT, we should detect be able to detect 65535/65536ths or 99.998 percent of all possible errors [3].

There is no technique which we can use to absolutely guarantee detection of any error; but we can minimize undetected errors at reasonable cost. Other error-detection techniques are available, such as checksum or voting, but these have poorer error-detection capabilities. For example, the single-byte checksum (used in the original version of XMODEM) appears to be about 99.29 percent accurate [4], which seems pretty good. But for a single additional byte, the CRC technique is about 460 times less likely to let an error pass undetected. In practice, the difference is much greater because the CRC will detect all cases of just a few errors, and these cases are most common. The cost is a 2-byte CRC value in every block. For example, the XMODEM protocol sends data in 128-byte blocks; these blocks can be CRC error-checked with an additional two bytes--an error-check overhead of about 1.5 percent [5].

Polynomial Arithmetic

The CRC performs its magic using polynomials modulo two arithmetic. Polynomial arithmetic mod 2 allows an efficient implementation of a form of division that is fast, easy to implement, and sufficient for the purposes of error detection. (This scheme is not particularly useful for the division of common numbers). Polynomial arithmetic mod 2 differs slightly from normal computer arithmetic, and is generally the most confusing part of the CRC.

A polynomial is a value expressed in a particular algebraic form, that of: A[n]*X^n + A[n-1]*X^n-1 + . . . + A[1]*X + A[0] (or AnXn + An-1Xn-1 + . . . + A1X + A0).

Our common number system is an implied polynomial of base 10: Each digit means that digit is multiplied by the associated power of 10. The base 2 or binary system of numeration is also a form of the general polynomial concept. When we see a number, we think of it as a single value; we mentally perform the polynomial evaluation in the assumed base to get a single result. On the other hand, formal polynomials are considered to be a list of multiple separate units, and the existence or evaluation of an ultimate single value for the polynomial may not be important.

Because decimal arithmetic uses constant-base polynomials, all of us already know how to do polynomial arithmetic in a constant base (10); however, the polynomials used in CRC calculations are polynomials modulo two. By modulo 2 we mean that a digit can have only values 0 and 1. Of course, this is always the case with binary values, so one might well wonder what all the mumbo-jumbo is about. The difference is this: A modulo polynomial has no carry operation between places [6]; each place is computed separately. We perform mod 2 operations logically, bit by bit; in mod 2, the addition operation is a logical exclusive-OR of the values, and mod 2 subtraction is exactly the same (exclusive-OR) operation.

Modulo arithmetic is used for CRC's because of its simplicity: Modulo arithmetic does not require carry or borrow operations. In computing hardware, the carry circuitry is a major part of arithmetic computation, and is a major contributor to speed limitations. Of course, since we have both subtraction and exclusive-OR instructions available in most computer instruction sets, this particular advantage is less important for software implementations of CRC. Nevertheless, the simplicity of modulo arithmetic allows several different software approaches not available in our conventional arithmetic. Note that the modulo-type operations available in programming languages (e.g., the Pascal MOD operator), operate on entire numbers rather than individual bits or places.

A polynomial division mod 2 is very similar to common binary division, except that we perform a logical exclusive-OR operation instead of a binary subtraction. Similarly, because "greater than" and "less than" are meaningless in modulo arithmetic, we can replace these operators by performing the exclusive-OR operation is the high bit is set or 1, driving the high part of the dividend to zeros.

We can implement a polynomial division as follows: A polynomial division register of a length corresponding to the remainder produced by the polynomial to be used is set up (see Figure 1, below) [7]. Each element of the register should be able to hold the maximum modulo value; in "mod 2," a single bit suffices. (Note that the hardware diagrams are intended only as examples; very short CRC's are of limited practical use, and there are better ways to do the job.)

```
    Polynomial = x^5 + x^4 + x^2 + 1 = 110101

    x^4         x^3     x^2         x^1     x^0
   +---+       +---+   +---+       +---+   +---+
+<-|Q D|<-XOR<-|Q D|<--|Q D|<-XOR<-|Q D|<--|Q D|<-XOR<- dn
|  |   |   ^   |   |   |   |   ^   |   |   |   |   ^
|  +---+   |   +---+   +---+   |   +---+   +---+   |
|          |                   |                   |
+----------+-------------------+-------------------+
```

FIGURE 1. Polynomial Divide Hardware for a 4 bit CRC

The register is cleared, then the data are shifted into the register from the right; each shift is a polynomial multiplication. Each shift also shifts a bit out of the register from the most significant bit (MSb). We know that the register value will exceed our representation when the shifted-out bit is logical 1, so we arrange to perform our polynomial subtraction" when this happens; that is, when we shift out a 1, we exclusive-OR the polynomial with the value in the register. Because our polynomial (the magic number) always contains a high-order bit, which always forces the shifted-out bit back to a logical 0, we need not actually operate on the high-order bit. So only zeros shift out, keeping the mod 2 polynomial remainder in the register.

This bit-level hardware process is easily simulated. Turbo Pascal algorithms for the simulation are shown in Listing One, page 76. Software simulation has the advantage of a fast and easy investigation of an algorithm, allowing quick changes to try out various forms of optimization. The program produces a "trace" of the execution, showing the step-by-step operation.

The polynomial division register does not hold the desired remainder until the place containing the last data bit has been shifted out of the register. To do this, a zero data bit be shifted in for each bit of the register. In the case of CRC-CCITT, 16 bits (2 bytes) of zeros need to be appended to the data. After entering the zero bits, the result in the polynomial division register is the CRC result. The common implementation of XMODEM requires these two trailing bytes.

The CRC result can be obtained without shifting in the two zero bytes by rearranging the CRC register and feeding the data in at the top end of the system (see Figure 2, below). By shifting the CRC register we can shift zeros in from the right. The data bit will be compared to the MSb in the CRC register, and only if they differ will the polynomial be subtracted. As before, this acts to keep the full remainder in the register; however, the remainder is now correct after each bit, and requires no trailing zeros.

```
       Polynomial = x^5 + x^4 + x^2 + 1 = 110101

       dn   x^4         x^3     x^2         x^1     x^0
       v   +---+       +---+   +---+       +---+   +---+
   +<-XOR<-|Q D|<-XOR<-|Q D|<--|Q D|<-XOR<-|Q D|<--|Q D|<-+
   |       |   |   ^   |   |   |   |   ^   |   |   |   |  |
   |       +---+   |   +---+   +---+   |   +---+   +---+  |
   |               |                   |                  |
   +---------------+-------------------+------------------+
```

FIGURE 2. CRC Hardware

A simulation of this immediate-result algorithm (called, for lack of a better name, the CRC algorithm) is also given in Listing One, for comparison to polynomial division. Notice that both the polynomial division and CRC algorithms come up with the same remainder (or CRC value), but the CRC version does it faster and with more consistent logic.

Faster CRC's in Software

The bit-by-bit form of the CRC algorithm can be, and often is, directly simulated in software. The shifting and looping required by this approach can be reduced in several ways. Both byte-oriented [8] and table-oriented [9] algorithms have been available in the technical literature for a number of years. Table-oriented algorithms may (or may not) produce somewhat higher speed, at the expense of a sizable table of constants that generally must be initialized before use. Examples of the various forms of CRC algorithms are given in Listing Two (page 78).

We can speed up the algorithm even more by precomputing the CRC for all possible combinations of a 16-bit CRC and a data byte and then saving the results. Done naively, this would be a transformation of 24 bits (16 bits of the previous register, and 8 bits of data) into 16 bits. This approach would thus require 2^25 Bytes (about 34 Megabytes) of look-up table. In order to make the table approach practical, we must find a way to reduce the size of the table.

If we examine the CRC hardware, we notice that the current data bit is always combined with the current MSb of the CRC register. When we compute a whole byte CRC, we end up combining the whole data byte with the high-byte of the CRC. We can precompute the exclusive-OR of the data byte and the high byte of the CRC register (this is a single operation in software), yielding a single byte we can call the combined term or the combination value.

For the common 16-bit CRC's, it turns out that the CRC register changes in patterns which are directly related to the combination value. Thus, it is possible to pre-compute the CRC changes for all 256 possible combination values. Then, when we need to do a CRC, we can use the 1-byte combination value to look up a corresponding 2-byte result, then use that result to correctly change the CRC register. As one might expect, the required change is simply a 2-byte exclusive-OR operation.

To generate the data for the lookup table, we need only generate the 2-byte CRC result for all 256 possible data bytes, given an "all zeros" starting CRC register. Each result has a 1 for those bits in the CRC register that are changed by a particular combination code. We can use a nontable implementation of the CRC to compute the table values.

This approach to generating a table of CRC values thus requires a 512-byte lookup table. We must fill the table with the correct data in an initialization step and perform a few more run-time operations than the straight lookup process requires (compute the combination value, look up the result, then apply the result to the CRC register and compute the new CRC value).

Another variations that is faster than the original bit-by-bit approach and that also eliminates the look-up storage of the table approach is the bytewide shifting algorithm. A bytewide approach eliminates seven bit-by-bit test-and-jump operations which are a significant overhead in the bit-by-bit version, and also takes advantage of fast shift and parallel-logic operations available on most processors (as well as some high-level languages such as Turbo Pascal or C).

First we need some algebra: By giving each CRC register bit and each data bit a separate symbol, we can express the result of a CRC operation symbolically. Each bit of the CRC register will be represented by a formula showing all the data and original CRC bits which affect that bit in the result. If we take the exclusive-OR of the bits specified by the formula, we can directly compute any bit of the CRC result.

In order to generate the formulas for each bit of the CRC register, we create an algebraic analog of the shifting and combining process of the bit-by-bit CRC algorithm. Instead of shifting bit values (as in a normal shift register), we instead move the whole symbolic formula for each bit to the next higher bit position. Instead of actually performing an exclusive-OR operation, we concatenate the formula for the data bit to each of the affected bits in the CRC register, with a symbol indicating an exclusive-OR operation. If ever we find that we have two identical variables in any one formula, we can cancel and eliminate them both (because anything exclusive-ORed with itself is zero, and zero exclusive-ORed with any value is just that value).

After symbolically processing a whole byte of data, and eliminating common terms, we come up with a symbolic representation for each bit of the result. By factoring this expression into convenient computer operations, a program is obtained which utilizes the bit parallelism available in software.

CRC Deviations

More improvement is possible. We have previously assumed that the CRC register is cleared before starting the computation, and also that we specifically compare the stored (or transmitted) CRC value to the current CRC result. These assumptions are discarded in protocols other than XMODEM [10].

When a CRC register is contains only zeros, processing a zero data bit does not change the CRC remainder. So, if the CRC register is clear, and extraneous zero bits do occur, these data errors will not be detected. For this reason, most current CRC protocols initialize the CRC register to all 1's before they start the computation, allowing the detection of extraneous leading zeros.

We can also eliminate the need to detect the separate CRC field at the end of a data block. If the CRC result is simply attached to the end of the data, the receiving CRC register will clear itself automatically if there is no error; that is, each bit of the stored or transmitted CRC value should cancel the similar bit in the CRC register. Although of minor importance for software implementations, this is a reasonable simplification for hardware CRC devices because it allows the exact same hardware to be used regardless of block length.

When the CRC is appended to the end of the data (thus eliminating the need to detect it as a separate field), and if bit-level CRC hardware is also to be supported, CRC software may need to use data in reverse bit order. This is because bit-level CRC hardware works on data after it has been serialized, and data is traditionally serialized LSb-first. That is, the parallel-to-serial conversion in an asynchronous serial device sends the rightmost bit of a character first and the leftmost bit last. The bit-level CRC hardware has little choice but to treat the resulting data stream as a single larger number; but that data-stream has its byte-level bit-order changed from our usual expectations.

If a MSb-leftmost CRC routine is to be compatible with bit-level CRC hardware, it may be necessary to reverse the bit order of every data byte (before each is processed or serialized) and also the CRC remainder bytes (after the block ends). Bit-order reversal can be done in software, hardware, or both. Alternately, the CRC algorithm could be constructed so as to use and hold MSb-rightmost data.

In strictly software CRC implementations, however, we work on data before it is serialized and after it is recovered and we trust any serialization that occurs to be transparent. We can thus afford to treat data as a single large value, MSb-leftmost, with MSb-leftmost bytes and a similar MSb-leftmost CRC remainder appended on the right. This arrangement is most consistent with both the theory and our numerical conventions, and is the form used by XMODEM. The CRC routines shown in this article use MSb-leftmost data and keep the result also in MSb-leftmost format.

If we arrange to verify the CRC by processing the CRC result as data, we again fall prey to extraneous zero data bits. In order detect such errors, we arrange for the CRC register to take on a unique nonzero value in the event of no error. By some quirk of the algebra, it turns out that if we transmit the complement of the CRC result and then CRC-process that as data upon reception, the CRC register will contain a unique nonzero value depending only upon the CRC polynomial (and the occurrence of no errors). This is the scheme now used by most CRC protocols, and the magic remainder for CRC-CCITT is $1D0F (hex).

Actual CRC Implementations

I constructed several CRC implementations for speed and size comparisons (see Listing Two). The CRC-CCITT polynomial was used, since this is the polynomial used in XMODEM, as well as many other data communication uses. I used Turbo Pascal although the code could obviously be rewritten in C. A couple of the operations used are Turbo Pascal extensions: Swap() is an INTEGER function that exchanges the high and low byte of an integer value; Lo() is an INTEGER function which selects only the low byte of an integer.

I used the Pascal Bit-by-Bit approach (a direct simulation of the hardware method) to provide a reference against which the other algorithms are compared. The Pascal Fast B-B-B is an improved bit form comparable to most high-level-language implementations of the XMODEM CRC, except that this version requires no trailing zeros to finish the calculation (and so is already faster than the usual version). The Pascal Byte version illustrates the improvement wrought from algebraic factoring; the Pascal Table version shows how a pre-computed table can simplify and speed execution-time operation. The Machine Code versions of Byte and Table show yet more improved speed. The different approaches illustrate various tradeoffs of speed, space, and specialization. The results (Table 1, left) show a range of almost two orders of magnitude in execution speed.

Each CRC implementation was made a Pascal PROCEDURE for easy testing and comparison. For validation, varying amounts of program code from main-memory were processed by each implementation. All algorithms achieved the same result. Several of these versions have been placed in an implementation of XMODEM with good results.

Time Tests

For the time tests, each implementation was executed 10,000 times under Turbo Pascal 3.01A on an 8088 in a Leading Edge PC with a 7.16 Megahertz (MHz) clock; the times would be 50% longer on an IBM PC. The time was taken automatically from MS DOS. Because the MS DOS timer ticks only about 18.2 times per second, this method is only precise within about 55 milliseconds (msec) on both the start and end of the timing interval. The large number of repetitions minimize the effect.

	10,000 Uses (secs)		1 Use (msec)	
	Procedure	In Line	Procedure	In Line
Pascal Bit-by-Bit	13.790	13.070	1.379	1.307
Pascal Fast B-B-B	7.310	6.590	0.731	0.659
Pascal Byte	2.150	1.430	0.215	0.143
Pascal Table	1.430	0.710	0.143	0.071
Machine Code Byte	1.050	0.330	0.105	0.033
Machine Code Table	0.890	0.170	0.089	0.017

TABLE 1.

The time reported as "10000 Uses" is real time decreased by the amount of time taken by 10,000 empty loops, thus giving us the time associated with the procedure call and execution, instead of also including the looping structure that we use only for the tests. The "In-Line" column decreases "10000 uses" by the time taken for 10,000 procedure calls and returns, giving the time for execution only.

Selection Criteria

The time necessary to process a byte (including the CRC operation, and whatever queuing operations and other tests need to be performed) should be less than the time it takes to receive a character. We could just accumulate the data in a block as it is received, then CRC-process the whole block, but this would add some delay, or latency, between receiving the last data byte and returning a response to the sender (ACK for good data, NAK for an error, in XMODEM). Some XMODEM implementations appear to use this method, giving the impression that the protocol or the CRC are responsible for the delay. Because fast CRC routines are obviously possible, it is hard to rationalize any latency at all [11].

The Pascal Byte version, which takes only a few lines of code and is machine-independent (under Turbo Pascal), may be suitable for speeds up to 9600 bps, and is a reasonable choice for most use. The Pascal Table version is a little faster, but the table generally must be initialized before use, either by using a different CRC version, or perhaps by reading the values in from a file. Alternately (in most languages) the table could be defined in the source code as a large body of constants.

The faster versions can generally benefit from being used in-line (that is, not as procedures) to avoid procedure call/return overhead, but this is also inconvenient, since each use would involve duplicating the same code in different places. The Machine Code Table version is shorter, and so would minimize the duplication penalty. The Pascal Table version can also be used in-line, because it takes a minimum amount of code. I use an Include file holding the Machine Code Byte version, then call the routine as a procedure; the resulting code is both small and fast.

Other Uses

Although this article has concentrated on CRC's in communications and data storage, CRC's can be used in many different applications involving error detection. Such applications include start-up verification of ROM code, load-time verification of RAM modules (as in the 6809 operating system OS9), and program and data correctness validation.

It should be noted that CRC polynomials are designed and constructed for use over data blocks of limited size; larger amounts of data will invalidate some of the expected properties (such as the guarantee of detecting any 2-bit errors). For 16-bit polynomials, the maximum designed data length is generally 2^15 - 1 bits, which is just one bit less than 4K bytes. Consequently, a 16-bit polynomial is probably not the best choice to produce a single result representing an entire file, or even to verify a single EROM device (which are now commonly 8K or more). For this reason, the OS9 polynomial is 24 bits long.

How To Learn More

A good introduction to CRC's can be found in the classic Error Correcting Codes, by Peterson and Weldon (Cambridge, Mass., MIT Press, 1972), but you can expect to do some serious math to understand it. A brief non-mathematical chapter on CRC error detection in data applications (with some good figures) is available in Technical Aspects of Data Communication, 2nd ed., by J. McNamara (Digital Equipment Corporation, Digital Press, 1982). The very brief section in Computer Networks by A. Tanenbaum is also fairly good.

Notes

1. The CRC does not require a fixed block size (though there is a built-in maximum), but some error-correcting protocols do. Larger amounts of data are simply partitioned into blocks that are considered separately.
2. Peterson, W. W. and D. T. Brown. 1961. Cyclic Codes for Error Detection." Proceedings of the IRE. January. 228-235.

3. Tanenbaum, A. 1981. Computer Networks. Prentice-Hall. 128-132.
4. Brooks, L. and J. Rasp. 1984. "How Accurate is Accurate?" DDJ. February. 27.
5. Error detection is only part of the requirements for a protocol. Other requirements include transmitting the data in blocks, numbering the blocks, and responding when a block has been received. The corresponding design decisions in XMODEM typically add yet another four bytes to each block transferred, for a required overhead of about 4.5 percent. This value can be, and often is, additionally degraded in implementation.
6. The general case of polynomial arithmetic, which allows a nonconstant base, generally makes carry operations (between terms) difficult.
7. It is common and traditional for the CRC register to be shown shifting right, which is the exact inverse of this author's analogy to binary division. Given our system of numeration, it seems reasonable to place the most significant digits of a value to the left, and it is then correct for the CRC register to be seen as shifting to the left.
8.
    Helness, K. 1974. "Implementation of a Parallel Cyclic Redundancy Check Generator." Computer Design. March. 91-96.
    Vasa, S. 1976. "Calculating an Error-Checking Character in Software." Computer Design. May. 190-192.
    Socha, H., et. al. 1979. "Letter to the editor." Computer Design. May. 6, 12.
    Kjelberg, I. 1985. "Letter to the editor." IEEE Micro. August. 4, 99.
9.
    Whiting, J. 1975. "An Efficient Software Method for Implementing Polynomial Error Detection Codes." Computer Design. March. 73-77.
    Perez, A. 1983. "Byte-wise CRC Calculations." IEEE Micro. June. 40-50.
    Schwaderer, D. 1985. "CRC Calculation." PC Tech Journal. 118-132.
10.
    McKee, H. 1975. "Improved CRC Technique Detects Erroneous Leading and Trailing 0's in Data Blocks." Computer Design. October. 102-106.
    Fortune, P. 1977. "Two-Step Procedure Improves CRC Mechanisms." Computer Design. November. 116-129.
11. Some protocols other than XMODEM allow subsequent blocks to be sent before a previous block is acknowledged, thus minimizing the latency problem.