CALCULATING CRCs
BY BITS AND BYTES

BY GREG MORSE

Use the XOR function to implement modulo 2 division when calculating cyclic redundancy checks

Recently I needed to implement the XMODEM cyclic redundancy check (CRC) option on my XCom9 modem program. Despite the many programs available for calculating CRCs, I had some difficulty understanding the math behind the calculations. The details became clear after much research and experimentation.

The starting point for all CRCs is fancy linear algebra. The CRC is defined in terms of message polynomials, generator polynomials, and so on. As Perez, Wismer, and Becker state in "Byte-wise CRC Calculations" in IEEE Micro, June 1983,

In a system employing CRCs, the message being transmitted is considered to be a binary polynomial \( M(x) \). It is first multiplied by \( x^8 \) and then divided (modulo 2) by an arbitrary generator polynomial \( G(x) \) of degree 4 which results in a quotient \( Q(x) \) and a remainder ...

It sounds confusing, but if you can understand how to apply the math, you can design more efficient programs and spot many erroneous ones.

THE "MATH-NESS" TO THE METHOD

The design of the polynomial \( G(x) \) is extremely complex. You need to pick one that produces CRCs that are good at detecting errors. Fortunately, many \( G(x) \)s exist already. Table 1 contains the two most common \( G(x) \)s in an 8-bit-byte environment.

Let's calculate the CRC for the letter T, 0101 0100 in binary. \( M(x) \) is the message as it is transmitted. You transmit a character's least significant bit (LSB) first, so \( M(x) \) becomes 00101010. Then you divide modulo 2 as shown in figure 1. (Modulo 2 means you use the XOR instruction instead of the normal add and subtract.) Work it through according to the process shown in figure 2. Note that the CRC result is given in reverse order, that is, most significant bit (MSB) on the right, LSB on the left.

BIT-ORIENTED ALGORITHMS

Using the long-division approach, if you had only a single zero bit to send, you would get the result shown in figure 3. If you had two bits to send, a zero and then a one, long division would produce the result shown in figure 4. The first remainder in figure 4 is the same as the first remainder in figure 3 except that its LSB has been XORed with 1.

Greg Morse (10871 Roseland Gate, Richmond, BC, Canada V7A 2R1) is an engineer with the British Columbia Telephone Company. He has 20 years experience with computers and 10 with data communications. Greg has B.A.Sc. and M.A.Sc. degrees in electrical engineering.
CALCULATING CRCs

Table 1: The most common CRCs (generator polynomials) in an 8-bit-byte environment. You can code the CRC as a 17-bit binary, or hexadecimal, number.

<table>
<thead>
<tr>
<th>CRC</th>
<th>Polynomial</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-16 (Bisynchronous)</td>
<td>(X^{16} + X^{15} + X^2 + X + 1)</td>
</tr>
<tr>
<td>CRC-CCITT (X.25, XMODEM)</td>
<td>(X^{16} + X^{12} + X^5 + 1)</td>
</tr>
<tr>
<td>CRC-8 (SDLC, X.25, XMODEM)</td>
<td>(X^8 + X^7 + X^5 + 1)</td>
</tr>
</tbody>
</table>

Figure 1: The initial modulo 2 division used to calculate the CRC for the letter T. Note that the letter is given LSB first; in other words, it is reversed.

The second data bit. This similarity will always be true because of the way the \(X^i/X^j\) polynomial is built. This observation leads to the following bit-by-bit algorithm for calculating the CRC:

1. Write down the first data bit (zero or one) to be transmitted.
2. Write down 16 zeros to its right.
3. Divide the 17-bit number by the 17-bit CRC polynomial using XOR instead of subtraction. Make a note of the remainder, which is the CRC.
4. Get the next data bit.
5. XOR this bit with the LSB (left-most bit of the CRC) in step 3.
6. Append a zero to the right-hand end of the result in step 5.
7. Divide the 17-bit number from step 6 by the 17-bit CRC polynomial. Use XOR instead of subtraction. The remainder is the CRC.
8. Repeat steps 4 through 7 until there are no more data bits. (You can replace steps 1 through 3 with a single step to initialize the CRC to zeros.)

Thus, you can calculate the CRC at a time, for any number of bits, with almost no extra calculation overhead. In summary, the steps involved in the long-division method are:

1. The message bits are written down in the order in which they are transmitted, from left to right, that is LSB on the left.
2. Sixteen zeros are appended to the right-hand end of the binary number formed in step 1.
3. The generating polynomial is written down MSB first, that is, on the left.
4. The division is done modulo 2, that is, using XOR instead of normal subtraction.
5. The CRC is the remainder after all data bits have been processed. The LSB is on the left.

Figure 2: The entire long-division calculation process used to derive the CRC for the letter T. The apparent subtractions in this process are XORs; therefore, there are no "errors." A "0" represents a zero that has no further part in the calculation and a "1" represents a zero that was "brought down" from the divided line. Therefore, the line \(00 1000 0101 0010\) represents a remainder of 00 1000 0101 0100 10. Note that the CRC is given with the LSB on the left. Thus, the CRC for the letter T = 0 54 is MSB first 0001 0100 1000 0101, or 14A1 hexadecimal.

Hardware Implementations

One disadvantage of the flowchart in figure 5 is that it requires a 17-bit register as a divider. This problem is solved by using a 16-bit counter, left shifted to the right by one bit. This approach is fast and uses less hardware than the divider approach.

Figure 5: A flowchart representing the steps involved in the long-division method of calculating a CRC.

(continued)
CALCULATING CRCs

Figure 6: The division process if Q = 1.

\[
\begin{array}{c|cccccccccccccccc}
Q = 1 & R1 & R2 & R3 & R4 & R5 & R6 & R7 & R8 & R9 & R10 & R11 & R12 & R13 & R14 & R15 & 0 \\
\hline
XOR & 0 & 0 & 0 & 0 & X12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{New CRC} & R0 & R1 & R2 & R3 & R4 & R5 & R6 & R7 & R8 & R9 & R10 & R11 & R12 & R13 & R14 & R15 & 0
\end{array}
\]

Figure 7: The division process if Q = 0.

\[
\begin{array}{c|cccccccccccccccc}
Q = 0 & R1 & R2 & R3 & R4 & R5 & R6 & R7 & R8 & R9 & R10 & R11 & R12 & R13 & R14 & R15 & 0 \\
\hline
XOR & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{New CRC} & R0 & R1 & R2 & R3 & R4 & R5 & R6 & R7 & R8 & R9 & R10 & R11 & R12 & R13 & R14 & R15 & 0
\end{array}
\]

Figure 8: The division process regardless of the value of Q.

\[
\begin{array}{c|cccccccccccccccc}
Q = 0 & R0 & R1 & R2 & R3 & R4 & R5 & R6 & R7 & R8 & R9 & R10 & R11 & R12 & R13 & R14 & R15 & 0 \\
\hline
XOR & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{New CRC} & R0 & R1 & R2 & R3 & R4 & R5 & R6 & R7 & R8 & R9 & R10 & R11 & R12 & R13 & R14 & R15 & 0
\end{array}
\]

Figure 9: The process shown in figure 8 translated into a hardware circuit using shift registers and XOR gates. (The boxes are stages to a shift register; the shift register is only 16 stages long.) Note that the XOR gate going into R15 is superfluous since Q XOR 0 is always Q. Note also that in the long-division method, the calculations are done before "bringing down" the next zero bit. Similarly, here, the XORs are performed before the shift.

Figure 10: The reverse drawing of figure 9: the LSB is on the right instead of on the left.

CALCULATING CRCs

later. However, after the division is complete, the least bit will always be a zero. Either it was a zero to begin with, or if it was a one, it was XORed with the X bits of the polynomial, which is also a one, producing a zero. Thus, you can XOR with only the 16 LSBs of the polynomial. In the case of the CRC-CCITT polynomial, X^16 + X^12 + X^5 + 1 or 0001 0000 0010 0001 in binary or 1 0112 in hexadecimal, you can XOR with 0 1021.

Let the bits of the CRC register be R0, R1, up to R15; let the data bits be D0 up to D7; let the polynomial bits be X0 up to X15; and define Q = D XOR R0; that is, set Q equal to the current data bit, D, XORed with R0. In every case, the LSB is bit D.

Then, if Q = 1, the division process looks like that in figure 6. If Q = 0, then the division process reduces to that in figure 7. But when Q = 1, the new R15 is always 1, and you XOR R4 and R11 with X12 and X5, which are 1. If Q = 0, then the new R15 is zero and you XOR R4 and R11 (and all other Rn) with zero.

You can combine the two cases as shown in figure 6. Turning that process into a circuit using shift registers and XOR gates (see figure 9) is straightforward. You can also draw the circuit so that the LSB (RO) is on the right instead of on the left.

// for whole BLK
/* XOR CRC with 0x0e01 for each char */
bytecrc(bufptr, crcres, count) unsigned char *bufptr, *crcres; count; /* defines POLY 0x8408 */
BLKCRC(bufptr, crcres, count) unsigned char *bufptr, *crcres; count;

Listing 1: CCITT, the CCITT routine for calculating CRCs in C source code.

/* Straightforward, non-optimized CCITT-CRC routine */
/* MSB of integer is MSB of CRC result */
#define POLY 0x8408

/* BLKCRC(bufptr, crcres, count) */
unsigned char *bufptr, *crcres, count; /* defines POLY 0x8408 */

CRe starts at a shift register; the shift register is only 16 stages long. Note that the XOR gate going into R15 is superfluous since Q XOR 0 is always 0. Note also that in the long-division method, the calculations are done before "bringing down" the next zero bit. Similarly, here, the XORs are performed before the shift.

BIT8-BIT SOFTWARE ALGORITHMS

When you do the calculations in software, you don't have to do the XOR before the shift. The important thing in software is to avoid having to deal with 17-bit values that don't fit into a variable. A software routine also doesn't need to use the X = D XOR R0 result to drive a gate. You can program the polynomial directly into the code as a constant.

You can now derive a software routine based on figure 10 (or figure 9) in which the order of storing the bits of the CRC is reversed. If you assume

that the CRC result is kept in a 16-bit integer with R15 the MSB of the CRC in the high (least) bit position, then if Q = 1, you first shift the previous CRC to the right as in figure 11. In terms of a high-level language, you shift the CRC right by one, discarding the LSB and XOR with 0x8408. By testing Q first and then doing the shift before the XOR, you avoid the need for a 17-bit register. Because of the way in which the CRC is stored in the variable, the XOR is done with 0x8408 rather than 0x1021, as you might expect. This follows directly from the

\[
C_0 = X \times \left( \frac{C_1}{X} \right)
\]

In every case, the LSB is bit D.

Then, if Q = 1, the division process looks like that in figure 6. If Q = 0, then the division process reduces to that in figure 7. But when Q = 1, the new R15 is always 1, and you XOR R4 and R11 with X12 and X5, which are 1. If Q = 0, then the new R15 is zero and you XOR R4 and R11 (and all other Rn) with zero.

You can combine the two cases as shown in figure 6. Turning that process into a circuit using shift registers and XOR gates (see figure 9) is straightforward. You can also draw the circuit so that the LSB (RO) is on the right instead of on the left.

Both diagrams are convenient depending on the kind of software algorithm to be derived. Note that in hardware the XOR is done before the shift because of the way flip-flops work. The important point is that the new R10 = old R11 XOR (old R11 XOR new data bit).

BIT8-BIT SOFTWARE ALGORITHMS

When you do the calculations in software, you don't have to do the XOR before the shift. The important thing in software is to avoid having to deal with 17-bit values that don't fit into a variable. A software routine also doesn't need to use the X = D XOR R0 result to drive a gate. You can program the polynomial directly into the code as a constant.

You can now derive a software routine based on figure 10 (or figure 9) in which the order of storing the bits of the CRC is reversed. If you assume

that the CRC result is kept in a 16-bit integer with R15 the MSB of the CRC in the high (least) bit position, then if Q = 1, you first shift the previous CRC to the right as in figure 11. In terms of a high-level language, you shift the CRC right by one, discarding the LSB and XOR with 0x8408. By testing Q first and then doing the shift before the XOR, you avoid the need for a 17-bit register. Because of the way in which the CRC is stored in the variable, the XOR is done with 0x8408 rather than 0x1021, as you might expect. This follows directly from the

\[
C_0 = X \times \left( \frac{C_1}{X} \right)
\]
CALCULATING CRCs

Generally speaking, if you process data LSB first, you can store the CRC with its MSB in the MSB position of the integer variable. Listing 1 contains the implementation of a CRC calculation based on the CRC in a standard way with one exception: initializing the CRC. This program implements the CRCs used by IBM in the SDLC protocol and CCITT in the X.25 and HDLC protocols. For example, you can add or delete any number of zero bits to or from the beginning of the block without affecting the CRC. The CRC is initialized to zeros at the beginning of the block without affecting the CRC. Furthermore, since the CRC is a cyclic code, any error, such as a clock slippage, that deletes a bit at the beginning of the block and inserts the same bit at the end of the block (the last bit of the CRC) will not affect the CRC. For these reasons the CRCs used by IBM in the SDLC protocol and CCITT in the X.25 and HDLC protocols specify the following:

1. All bits of a block are protected by the CRC.
2. The data is sent LSB first. The CRC is calculated on bits as they are sent.
3. The CRC is initialized to all ones. This allows detection of any missed or inserted zero bits at the beginning of a block. (Missed or inserted ones are still detected.)
4. The CRC's complement of the

Listing 2: XMODEM, the XMODEM routine for calculating CRCs in C source code.

```c
int calcrc(ptr, count)
char *ptr;
int count;

unsigned int crc;
int i;
while (--count > 0) {
    if (crc & 0x8000) {
        crc = crc >> 1;
        for (i=0; i<8; i++)
            if (crc & 0x8000) {
                crc = crc ^ 0xA001;
                break;
            }
        else
            return (crc ^ 0xFFFF);
    } else
        return (crc ^ 0xFFFF);
    }
}
```

Some bit-oriented CRC routine adapted from XMODEM protocol reference

The single best way to turn your PC-AT into a multi-user system.

The single best way to turn your PC-AT into a multi-user system.

Introducing the Wyse WY-60
Now there's a perfectly compatible, reliable, economical Wyse way to get multi-user mileage from your PC-AT. Wyse WY-60 terminals give you complete compatibility for your IBM Personal Computer AT systems, right down to the exact keyboard layout, character set and display features.

The only thing different is how much cleaner and more readable your information is with the WY-60's high resolution flat, non-glare, 14" swivel screen. Multiple display formats go up to 132 columns and 44 lines on one screen, to get the most out of applications such as Multiplan and WordStar.

And a 512-character downloadable soft font is also there when you need mathematical symbols or customized character sets.

The adjustable arm is optional, and you can choose a green, white or amber screen.

No wonder we ship more terminals than anybody but IBM.*

Call toll-free or write, today, for more information. Wyse Technology, Attn: Marcom Department 69-AT, 3751 N. First St., San Jose, California 95134.

Call 1-800-GET-WYSE

Wyse

YOU NEVER REGRET A WYSE DECISION.
CALCULATING CRCs

**Figure 12:** The contents of the shift register at the beginning of bit per byte SDLC calculations.

<table>
<thead>
<tr>
<th>Bit</th>
<th>*5 14 13 12 11 *10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R15 R14 R13 R12 R11 R10 R9 R8 R7 R6 R5 R4 R3 R2 R1 R0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 13:** The contents of the shift register after the first shift. Note that all entries in a column are XOR-ed together.

<table>
<thead>
<tr>
<th>Bit</th>
<th>*5 14 13 12 11 *10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift 1</td>
<td>D0 R15 R14 R13 R12 R11 R10 R9 R8 R7 R6 R5 R4 R3 R2 R1 R0</td>
</tr>
<tr>
<td>Shift 1</td>
<td>R15 R14 R13 R12 R11 R10 R9 R8 R7 R6 R5 R4 R3 R2 R1 R0</td>
</tr>
</tbody>
</table>

**Figure 14:** The same as figure 13 but with the abbreviation TO for DO XOR RO.

**Figure 15:** The contents of the shift register after three more right shifts. TO represents D0 XOR RO. R1 represents D1 XOR R1, and so on.

**Figure 16:** The contents of the shift register after the fifth shift.

**Figure 17:** The contents of the shift register after the eighth shift.

CRC is transmitted rather than the CRC itself. This allows detection of slippage-type errors.

1. The CRC is sent LSB first.
2. The polynomial used is \( X^4 + X^3 + X^2 + X^1 + 1 \).
3. The CRC is calculated on the data MSB first.
4. The CRC is initialized to zeros.
5. The order of transmission is high byte of CRC then low byte. Since UARTs transmit LSB first, this is equivalent to LSB of high byte through MSB of high byte, then LSB of low byte through MSB of low byte.
6. The polynomial used is \( X^4 + X^3 + X^2 + X^1 + 1 \).

The CCITT method is easily implemented in a chip using shift registers and XOR gates, whereas the XMODEM method is not easily realized in hardware. To check an incoming data block, you have two options. You can calculate the CRC on all the protected bits only, omitting the CRC bits and compare the calculated value to the received value. Or you can calculate the CRC on all the protected bits and the CRC itself and then compare the result to a known constant. If the first method is adopted in the CCITT case, the one's complement of the calculated value must be compared to the received CRC. In the XMODEM case, the comparison is direct. If the second method is adopted. In the XMODEM case the known constant is zero, while in the CCITT case it is 0 F08h (the high bit is on the left, i.e. 1). No one's-complementing is required.

### Byte-oriented Software Implementations

The next step is to derive routines to calculate the CRC a whole byte in a time rather than the bit by bit. This approach was first proposed by Perez, Wiemer, and Becker. The motivation is that an 8-bit microprocessor is not limited to single-bit XORs but can do them 8 bits at a time. The basic approach is to work from the hardware diagram (figure 9 or 10) and see what the CRC register would look like after 8 bits have been calculated. Bytewise SDLC calculations in figure 10 you can see that you have the content of figure 12 in the shift register at the beginning of calculations. Then you take the LSB of data, D0, and XOR it with RO (DO XOR RO). After the right shift, the new R15 is D0 XOR RO, the new R10 is D1 XOR RO, and the new R5 is R4 XOR DO (DO XOR RO). See figure 13.

The combinations D0 XOR RO, D1 XOR R1, D2 XOR R2, and so on, occur frequently, so let's abbreviate them as TO = DO XOR RO, TI = D1 XOR R1, and so on. Figure 13 can now be rewritten as in figure 14. If you proceed in this fashion for three more shifts, you get figure 15.

To do the XOR on D4, you must use the content of the LSB of the shift register, which is now R4 XOR DO XOR RO. The result is R4 XOR DO XOR RO, or in shorter form, T4 XOR TO. The result after the fifth shift is shown in figure 16; after the eighth shift, in figure 17.

The tedious part is done. Now you want to write a program that will produce the same result when you're working with a byte of data as you would get from the shift register after eight shifts. The emphasis in this process will be on speed. If you make it too general-purpose, a different choice of polynomial will lead to a completely different program. Also, for speed, it makes sense to code the routine in assembly language.

When working with 8-bit microprocessors, it is convenient to define the 8-bit quantities found in table 2. If you study figure 17, you will see that the combination

\[ T7 \times T6 \times T5 \times T4 \times T3 \times T2 \times T1 \]

occurs several times. Let's call this term U = U7 U6 U5 U4. If you rewrite figure 17 with these substitutions, you can derive the results of figure 18.

Further optimizations become apparent as you write the code. It is convenient to implement the SDLC algorithm in the program SDLC.ASM because of the built-in check that calculating a CRC on a 'received' block provides: that is, the result should always be 0 F08h. The innermost loop of SDLC.ASM takes only 8 cycles per byte including seven overhead cycles to check for end of buffer if a 2-MHz 6809 were dedicated to CRC calculations, this would result in a through-

### Table 2: Some convenient abbreviations for various 8-bit quantities used in CRC calculations in an 8-bit microcomputer environment.

<table>
<thead>
<tr>
<th>Test</th>
<th>CRC</th>
<th>Test</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>17B26</td>
<td>14A1</td>
<td>THE</td>
<td>448E</td>
</tr>
<tr>
<td>THE</td>
<td>26</td>
<td>QUICK.BROWN.FOX</td>
<td>0123456789</td>
</tr>
<tr>
<td>DC1</td>
<td>T0C5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 18:** The same as figure 17 but using the abbreviations given in table 2.

1. Line 1 is CRCHI moved into CRCLO: line 2 is the high nibble of U and the low nibble of T. line 3 is the line 2 byte shifted left by 3 bits; line 4 is U shifted right by 4 bits.

### Table 3: Some test cases provided for comparison if you want to write your own routine.

<table>
<thead>
<tr>
<th>Test</th>
<th>SDLC</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>17B26</td>
<td>14A1</td>
<td>THE</td>
</tr>
<tr>
<td>THE</td>
<td>448E</td>
<td>T0C5</td>
</tr>
<tr>
<td>DC1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 19:** The contents of the shift register at the beginning of byte per byte XMODEM calculations.

<table>
<thead>
<tr>
<th>Test</th>
<th>SDLC</th>
<th>CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>17B26</td>
<td>14A1</td>
<td>THE</td>
</tr>
<tr>
<td>THE</td>
<td>448E</td>
<td>T0C5</td>
</tr>
<tr>
<td>DC1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 20:** The contents of the shift register after the XDR with the data byte.
Figure 21: The contents of the shift register after processing the first data byte and CRC calculation at TB.

Figure 22: The contents of the shift register after four shifts. In this state, the contents of the register are copied to RB.

Figure 23: After the XOR with the data byte. You now can calculate the CRC by applying the XMODM CRC algorithm shown in figure 24. After the last shift, you get the result shown in figure 25. If your mode is 64, the result is copied to RC, otherwise to RB and the contents of the shift register become zero.

Figure 24: The XMODM CRC algorithm is shown in this figure. After the XOR function is applied to the data byte, the contents of the shift register are copied to RB. Now you can calculate the CRC by applying the XMODM CRC algorithm shown in figure 25. After the last shift, you get the result shown in figure 26. If your mode is 64, the result is copied to RC, otherwise to RB.

Figure 25: After the last shift, the contents of the shift register are copied to RB. Now you can calculate the CRC by applying the XMODM CRC algorithm shown in this figure. After the last shift, you get the result shown in figure 26. If your mode is 64, the result is copied to RC, otherwise to RB.

Figure 26: After the last shift, the contents of the shift register are copied to RB. Now you can calculate the CRC by applying the XMODM CRC algorithm shown in this figure. After the last shift, you get the result shown in figure 27. If your mode is 64, the result is copied to RC, otherwise to RB.