A cyclic redundancy code can be calculated on bytes instead of bits. One byte-oriented method reduces calculation time by a factor of almost four.

Byte-wise CRC Calculations

Aram Perez
Wismer & Becker

A method that is commonly used to ensure the integrity of the messages in data communications is a cyclic redundancy code, or CRC. A CRC is usually calculated automatically in hardware by means of a bit-wise method. It can also be calculated in software by emulating this method. Here, we derive a byte-wise algorithm for calculating CRC in software that is four times faster than the usual bit-wise software method. The idea for this algorithm came from a table look-up algorithm by Lee. The method described here eliminates the table and takes no more space than the slower bit-wise method.

CRC background and theory

When digital messages are transmitted and received over telephone or radio channels, some errors can be expected to appear. Errors occur because of interference between channels, fading of signals, atmospheric conditions, and other sources of noise. Some method is needed to detect when the message received is not the same as that transmitted. Commonly used methods of detecting errors include checksums, parity checks, longitudinal redundancy code, and cyclic redundancy code.

CRC is often used because it is easy to implement and it detects a large class of errors. For any given message, CRC will detect

- all one- or two-bit errors,
- all odd numbers of bit errors,
- all burst errors less than or equal to the degree of the polynomial used, and
- most burst errors greater than the degree of the polynomial used.

In a system employing CRC, the message being transmitted is considered to be a binary polynomial M(X). It is first multiplied by X^k and then divided by an arbitrary
generator polynomial $G(X)$ of degree $k$, which results in a
quotient $Q(X)$ and a remainder $R(X)/G(X)$. All arithmetic
is done in modulo 2. This process is shown in the following
equation, in which $\oplus$ is the sign for addition in modulo 2
arithmetic:

$$X^k M(X) \oplus R(X) = Q(X) \oplus G(X).$$

In modulo 2 arithmetic, the results of subtraction are
equivalent to the results of addition. By applying this
property and some simple algebra to the equation, we get

$$X^k M(X) \oplus R(X) = Q(X)G(X).$$

$R(X)$ will always be of degree $k$ or less.

The CRC algorithm calculates $R(X)$ and appends it to
the message being sent. Since $X^k M(X) \oplus R(X)$ equals
$Q(X)G(X)$, the original message with the CRC appended
will be evenly divisible by $G(X)$, if and only if no bits are
changed. At the receiving end, the received message (ori­
ginal message plus $R(X)$) is divided by the generator
polynomial $G(X)$. If the remainder is nonzero, it is as­
sumed that an error has occurred. If the remainder is zero,
it is assumed that no errors have occurred or that an error
has occurred but has gone undetected by the algorithm. A
list of commonly used generator polynomials is given in
Table 1.

The CRC-16 polynomial is a common standard used
around the world. (It is the polynomial used in the Bisync
protocol, for example.) SDLC—synchronous data link
control—is used by IBM and is the standard in Europe.
The CRC-12 polynomial is used with six-bit bytes. The
“reverse” polynomials are the same as the “forward”
polynomials, except that the data are taken in reverse
order. The LRC polynomials are used in longitudinal
calculations.

The rest of this article will be concerned only with the
CRC-16 polynomial, although any other polynomial can be
easily adapted by using the selected polynomial in the
derivation.

Since CRC arithmetic is done in modulo 2, it can be
easily implemented in hardware with shift registers and
exclusive-OR gates (Figure 1). Each flip-flop contains one
bit of the CRC register. Most software routines emulate
the hardware method, thus operating on one bit at a time.
Since most processors are not bit-oriented, the bit-wise
software approach requires lengthy periods of CPU time.
Given that many microprocessors are byte-oriented, an
algorithm to calculate CRC on a byte-by-byte basis would
be of great benefit.

**Algorithm derivation**

Since we want to calculate the CRC eight bits at a time,
we need an algorithm that will produce the same CRC
value as would occur after eight shifts of a bit-wise CRC
calculation. The following sections derive such an al­
gorithm. Table 2 shows the CRC register for each of eight
shifts. The notation used is as follows:

- Bits are numbered starting at 1, and bit 1 is the least
  significant bit.
- The “SH” column is the shift number.
- The “IN” column is the data in, with $M_i$ being the $i$th
  bit of the current byte of message $M(X)$.
- $R_i$ is the $i$th bit of the CRC register.
- $C_j$ is the $j$th bit of the initial CRC register, just before
  any shifts due to the current input byte.
- Vertical entries in the $R_i$ columns denote that the en­
tries are to be exclusive-ORed to form the contents of each $R_i$.

As can be seen in Table 2, the contents of the CRC register
after eight shifts are a function (exclusive-OR) of various
combinations of the input data byte and the previous con­
tents of the CRC register. The byte-wise algorithm must
produce these CRC register contents.

| Table 1. Commonly used generator polynomials. |
|-----------------|-----------------|-----------------|
| CRC-16          | $X^{16} + X^{15} + X^2 + 1$ |
| SDLC (IBM, CCITT)| $X^{16} + X^{12} + X^6 + 1$ |
| CRC-12          | $X^{12} + X^{11} + X^5 + X^4 + 1$ |
| CRC-16 REVERSE  | $X^{15} + X^{14} + X + 1$ |
| SDLC REVERSE    | $X^{16} + X^{11} + X^4 + 1$ |
| LRCC-16         | $X^8 + 1$        |
| LRCC-8          | $X^8 + 1$        |

| Figure 1. Hardware for CRC-16 calculation. |
The exclusive-OR function has the following properties, given here without proofs (\( \oplus \) means exclusive-OR):

- \( A \oplus B = B \oplus A \) (commutativity).
- \( A \oplus B \oplus C = A \oplus C \oplus B \) (associativity).
- \( A \oplus A = 0 \) (involution).
- \( A \oplus 0 = A \) (identity).

The use of these properties makes it possible to simplify the contents of each bit of the CRC register after eight shifts, as shown in Table 3.

By defining a function \( X_i \), we can further simplify the CRC register. The vector \( X \) is composed of \( X_i \)'s which in turn are the result of the exclusive-OR of the \( i \)th bit of the input data byte with the \( i \)th bit of the CRC register. The function \( X_i \) is defined as

\[
X_i = C_i \oplus M_i.
\]

For an eight-bit data byte, \( X \) is the result of exclusive-OR-ing the low-order byte of the CRC register with the input

---

**Table 2. CRC register after eight shifts.**

<table>
<thead>
<tr>
<th>SH</th>
<th>IN</th>
<th>CRC REGISTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C0</td>
<td>C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1</td>
</tr>
<tr>
<td>1</td>
<td>M1</td>
<td>C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1</td>
</tr>
<tr>
<td>2</td>
<td>M2</td>
<td>C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1</td>
</tr>
<tr>
<td>3</td>
<td>M3</td>
<td>C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1</td>
</tr>
<tr>
<td>4</td>
<td>M4</td>
<td>C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1</td>
</tr>
<tr>
<td>5</td>
<td>M5</td>
<td>C16 C15 C14 C13 C12 C11 C10 C9 C8 C7 C6 C5 C4 C3 C2 C1</td>
</tr>
<tr>
<td>Table 2 cont'd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>M₆</td>
<td>C₆</td>
</tr>
<tr>
<td></td>
<td>C₁₅</td>
<td>C₁₄</td>
</tr>
<tr>
<td>M₅</td>
<td>C₄</td>
<td>C₃</td>
</tr>
<tr>
<td>C₄</td>
<td>C₃</td>
<td>C₂</td>
</tr>
<tr>
<td>C₂</td>
<td>C₁</td>
<td>M₃</td>
</tr>
<tr>
<td>M₃</td>
<td>M₂</td>
<td>M₁</td>
</tr>
<tr>
<td>7</td>
<td>M₇</td>
<td>C₇</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₆</td>
<td>C₅</td>
<td>C₄</td>
</tr>
<tr>
<td>C₅</td>
<td>C₄</td>
<td>C₃</td>
</tr>
<tr>
<td>C₃</td>
<td>C₂</td>
<td>C₁</td>
</tr>
<tr>
<td>M₄</td>
<td>M₃</td>
<td>M₂</td>
</tr>
<tr>
<td>8</td>
<td>M₉</td>
<td>C₈</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M₈</td>
<td>C₇</td>
<td>C₆</td>
</tr>
<tr>
<td>C₇</td>
<td>C₆</td>
<td>C₅</td>
</tr>
<tr>
<td>C₅</td>
<td>C₄</td>
<td>C₃</td>
</tr>
<tr>
<td>M₅</td>
<td>M₄</td>
<td>M₃</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

June 1983
data byte. Table 4 shows the new simplified CRC register using $X_i$'s.

From Table 4 we can see that

- the high-order byte of the CRC register is dependent only on combinations of the exclusive-OR of the initial lower eight bits of the CRC register and the input data byte, and that
- the low-order byte of the CRC register is dependent on functions of the initial lower eight bits of the CRC register, the input data byte, and the initial high-order eight bits of the CRC register.

This leads to the conclusion that it is possible to shift the high-order byte into the low-order byte of the CRC register, discard the low-order byte, and then exclusive-

### Table 3.
Simplified CRC register after eight shifts.

| SH IN | R16 | R15 | R14 | R13 | R12 | R11 | R10 | R9  | R8  | R7  | R6  | R5  | R4  | R3  | R2  | R1  |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 8     | M8  | C8  | C7  | C6  | C6  | C5  | C4  | C3  | C2  | C1  | C0  | C0  | C0  | C0  | C0  |
|       |     | M7  | M6  | M6  | M5  | M5  | M4  | M3  | M3  | M1  | M0  | M0  | M0  | M0  | C0  |
|       | M7  | M6  | M5  | M4  | M3  | C3  | C2  | C2  | M2  | M2  | M1  | M1  | M1  | M1  | M1  |
|       | C6  | C5  | C4  | C4  | M4  | M4  | M3  | M3  | C3  | C3  | M2  | M2  | M2  | M2  | M1  |
|       | M3  | M2  | C2  | M2  | C2  | C2  | C1  | C1  | C1  | M2  | M1  | M1  | M1  | M1  | M1  |
|       | C3  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|       | M1  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

### Table 4.
CRC register after eight shifts, using $X_i$.

<table>
<thead>
<tr>
<th>SH IN</th>
<th>R16</th>
<th>R15</th>
<th>R14</th>
<th>R13</th>
<th>R12</th>
<th>R11</th>
<th>R10</th>
<th>R9</th>
<th>R8</th>
<th>R7</th>
<th>R6</th>
<th>R5</th>
<th>R4</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>M8</td>
<td>X8</td>
<td>X7</td>
<td>X6</td>
<td>X6</td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X7</td>
<td>X6</td>
<td>X5</td>
<td>X6</td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>X6</td>
<td>X5</td>
<td>X5</td>
<td>X4</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X3</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>X4</td>
<td>X4</td>
<td>X4</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>X3</td>
<td>X3</td>
<td>X3</td>
<td>X3</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>X2</td>
<td>X2</td>
<td>X2</td>
<td>X2</td>
<td>X2</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
<td>X1</td>
</tr>
<tr>
<td></td>
<td>X1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 5.
Final CRC register.

<table>
<thead>
<tr>
<th>SH IN</th>
<th>R16</th>
<th>R15</th>
<th>R14</th>
<th>R13</th>
<th>R12</th>
<th>R11</th>
<th>R10</th>
<th>R9</th>
<th>R8</th>
<th>R7</th>
<th>R6</th>
<th>R5</th>
<th>R4</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>M8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X8</td>
<td>X7</td>
<td>X6</td>
<td>X6</td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X7</td>
<td>X6</td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X6</td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X5</td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X4</td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X3</td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X2</td>
<td>X1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
OR some 16-bit word with the CRC register to get the new contents. This is demonstrated in Table 5, in which everything below the dotted line defines the 16-bit word that is needed.

Given the results shown in Table 5, it is possible to state the following algorithm:

1. Exclusive-OR the input byte with the low-order byte of the CRC register to get the X_i's.
2. Shift the CRC register eight bits to the right.
3. Calculate a value from the X_i's which will give the 16-bit value defined by everything below the dotted line in Table 5.
4. Exclusive-OR the CRC register with the calculated value.
5. Repeat Steps 1 to 4 for all the message bytes.

Since the value calculated in Step 3 is dependent only on X_8 through X_1, and since there are only 256 different combinations of X, it is clear that these values can be tabulated using X as an index. Thus, the algorithm can be restated as follows:

1. Exclusive-OR the input byte with the low-order byte of the CRC register to get X.
2. Shift the CRC register eight bits to the right.
3. Exclusive-OR the CRC register with the contents of the table, using X as an index.
4. Repeat Steps 1 to 3 for all the message bytes.

This algorithm is general since no a priori assumption of the polynomial is necessary. Only the entries in the table change if a different polynomial is used. Thus, it is possible to use only one program to calculate a number of different CRCs.

Table generation

A Fortran 77 program that will generate the table values needed for byte-wise CRC-16 calculations is given in Listing 1, reproduced at the end of this article. It prints the values in hexadecimal on a line printer. A modified version of the program was used to make the file "CRCTB." This file was then INCLUDED in the source file of the 8080/8085 implementation of the algorithm. The program may be easily changed to calculate the values for a different CRC polynomial. Table 6 contains the table for the CRC-16 polynomial. It contains all the possible values that result from exclusive-ORing the low-order byte of the CRC register with the incoming data byte and that are defined by everything below the dotted line in Table 5.

Implementation

Listing 2 (program "CRCT") is an 8080/8085 program that implements the byte-wise CRC algorithm using the table look-up method. Comparing this routine with a bit-
Table 7.
Comparison of CRC routines.

<table>
<thead>
<tr>
<th>NUMBER OF CPU CYCLES TO CALCULATE CRC-16</th>
<th>CRCT</th>
<th>CRCF</th>
<th>CRCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER OF BYTES OF MEMORY FOR ROUTINE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRCT</td>
<td>540</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRCF</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRCB</td>
<td>44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Acknowledgments

I would like to thank Dick Wallace, Leo Endres, Dick Huffman, Roger Melton, Ray Haider, and, in particular, Fred Jensen, all of Wismer & Becker, for their invaluable support in the preparation of this article.

Performance

We compared both CRC routines against a bit-wise serial routine (which we called "CRCB") and obtained the results shown in Table 7. As can be seen, CRCT is the fastest routine, nearly five times as fast as CRCB. But CRCB does need almost twelve times as much memory as CRCT. The surprise turned out to be CRCF. Needing one byte less of memory than CRCB, it is nearly four times faster than that routine.

Reference


For further reading


Listing 1. Table generator program.

```fortran
FTN77,L !FORTRAN 77 (HP-1000)  
PROGRAM CRCV  
C  
C THIS PROGRAM CALCULATES THE VALUES NECESSARY FOR  
C BYTE-WISE CRC-16 CALCULATIONS.  
C  
C IT PRINTS THE HEX EQUIVALENT OF THE VALUES ON THE LINE PRINTER.  
C  
INTEGER X8, X7, X6, X5, X4, X3, X2, X1, V(16), P2(4)  
DIMENSION IA(4), IHXASC(16) ! HEX TO ASCII TABLE  
DATA IHXASC /"0","1","2","3","4","5","6","7",  
+ "8","9","A","B","C","D","E","F"/  
DATA V /0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0/  
DATA P2 /1,2,4,8/  
C  
C OUT= 6  
C  
WRITE(IOU,"(" X Value")")  
WRITE(IOU,"(" == ------")")  
C  
IXH= 1  
IXL= 1  
C  
START THE CALCULATIONS  
C  
DO X8= 0,1  
DO X7= 0,1  
DO X6= 0,1  
DO X5= 0,1  
DO X4= 0,1  
DO X3= 0,1  
DO X2= 0,1  
DO X1= 0,1  
X= X8,XOR,X7,XOR,X6,XOR,X5,XOR,X4,XOR,X3,XOR,X2,XOR,X1  
V(16)= X  
V(15)= X7,XOR,X6,XOR,X5,XOR,X4,XOR,X3,XOR,X2,XOR,X1  
V(14)= X8,XOR,X7  
V(13)= X7,XOR,X6  
V(12)= X6,XOR,X5  
V(11)= X5,XOR,X4  
V(10)= X4,XOR,X3  
V(9)= X3,XOR,X2  
V(8)= X2,XOR,X1  
V(7)= X1  
V(1)= X  
DO I= 4,1,-1 ! CONVERT BINARY TO HEX  
L= 0  
K= 4*(I-1)```

June 1983
DO J= 1,4
   L= L+(P2(J)*V(K+J))
END DO
IA(I)= IHXASC(L+1) ! CONVERT HEX TO ASCII
END DO
IXHA= IHXASC(IXH)
IXLA= IHXASC(IXL)
WRITE(IOUT,30) IXHA, IXLA, (IA(I), I= 4,1,-1)
IXL= IXL+1
IF(IXL .EQ. 17) THEN
   IXL= 1
   IXH= IXH+1
ENDIF
END DO
END DO
END DO
END DO
END DO
END DO
C
C  ALL DONE!
C
30  FORMAT(1X,2A1,2X,4A1)
C
END

Listing 2. Table look-up CRC routine.

;------------------------------------ _ .----------
; CRCT
; CALCULATES CRC-16 A BYTE AT A TIME USING A
; TABLE LOOK-UP ALGORITHM.
;
; GIVEN: 
; B= BYTE COUNT
; HL= BUFFER ADDRESS
;
; RETURNS: 
; B= 0
; C= C
; DE= CRC-16
; HL= BUFFER ADDRESS+ BYTE COUNT
; A= ??
;
;------------------------------------
CRCT EQU $ ;BEGIN
LXI D,0 ;INITIALIZE CRC
CRCT0 EQU S ;REPEAT
MOV A,M ; GET DATA BYTE
INX H ; BUMP POINTER
PUSH B ; SAVE_COUNTER
PUSH H ; SAVE DATA_POINTER
XRA E ; XOR DATA AND LOW BYTE OF CRC TO GET 'X'
MOV C,A ; FORM INDEX INTO TABLE
MVI B,0
LXI H,TCRC16 ; POINT TO TABLE
DAD B
DAD B ; INDEX INTO THE TABLE
MOV A,D ; SHIFT CRC 8 BITS
XRA M ; XOR TABLE ENTRY
MOV E,A ; REG.E= LO BYTE CRC
INX H
MOV D,M ; REG.D= HI BYTE CRC
POP H ; RESTORE DATA POINTER
POP B ; RESTORE COUNTER
DCR B ; DECREMENT COUNTER
JNZ CRCTO ; UNTIL BYTE COUNT=O
RET ;END

TCRC16 EQU S ; CRC TABLE
INCLUDE &CRCTB ;TABLE PRODUCED BY FORTRAN PROGRAM)
;
END


;-----------------------------
; CRCF
; CALCULATES CRC-16 A BYTE AT A TIME CALCULATING THE
; VALUES IT NEEDS 'ON THE FLY'.
;
; GIVEN: 
; B= BYTE COUNT
; HL= BUFFER ADDRESS
;
; RETURNS: 
; B= O
; C= C
; DE= CRC16
; HL= BUFFER ADDRESS+ BYTE COUNT
; ALL OTHERS CHANGED
;
; STRATEGY (FOR GETTING VALUE):
;
; 1. XOR DATA BYTE WITH LOW BYTE CRC
; REG.A= X
; 2. COPY X IN REG.L
3. SHIFT X LEFT 1 BIT BY ADDING REG.A WITH ITSELF
   CY= X8, P= XX7
4. SAVE X8 AND XX7
5. XOR REG.A WITH REG.L TO GET R14 THROUGH R7
   REG.A=X.AXOR.(X.SHL.1)
6. SAVE REG.A IN REG.L
7. RESTORE CY AND P
   CY= X8, P= XX7
8. MAKE XX7 EQUAL TO 0 AND XX8 EQUAL TO XX7
9. IF XX7 ACTUALLY IS 1 MAKE XX7 AND XX8 EQUAL TO 1
10. IF X8 EQUALS 1 THEN COMPLEMENT XX8
11. SAVE XX8 AND XX7 IN REG.H
   REG.HL(BITS 9 - 0)= R16 THROUGH R7
12. SHIFT REG.A RIGHT 1 BIT TO GET XX8 IN BIT 0
13. SHIFT REG.HL LEFT 6 BITS
14. PUT XX8 (FROM REG.A) IN REG.L
   REG.HL= R16 THROUGH R1

---

CRCF EQU $ ;BEGIN
CRCF0 EQU $ ;REPEAT
   LXI D,0 ;INITIALIZE CRC
   MOV A,M ; GET DATA BYTE
   INX H ; BUMP POINTER
   PUSH H ; SAVE DATA POINTER
   XRA E ; STEP 1
   MOV L,A ; STEP 2
   ADD A ; STEP 3
   PUSH PSW ; STEP 4
   XRA L ; STEP 5
   MOV L,A ; STEP 6
   POP PSW ; STEP 7
   MVI A,0 ; STEP 8
   JPE CRCF1
   MVI A,011B ; STEP 9
CRCF1 EQU $ ;BEGIN
   JNC CRCF2
   XRI 010B ; STEP 10
CRCF2 EQU $ ;REPEAT
   MOV H,A ; STEP 11
   RAR ; STEP 12
   DAD H ; STEP 13
   DAD H
   DAD H
   DAD H
   DAD H
   DAD H
   DAD H
   DAD H
   ORA L ; STEP 14
   XOR HIGH ORDER CRC (IMPLICIT .SHR.8)
   MOV E,A ; REG.E= LO BYTE CRC
   MOV D,H ; REG.D= HI BYTE CRC
   POP H ; RESTORE DATA POINTER
   DCR B ; DECREMENT COUNTER
   JNZ CRCF0 ;UNTIL BYTE COUNT=0
   RET ;END