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.[1] 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

0272-1732/83/0600-0040$01.00

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 ⊕ is the sign for addition in modulo 2 arithmetic:

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

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 (original message plus R(X)) is divided by the generator polynomial G(X). If the remainder is nonzero, it is assumed 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 algorithm. 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 ith bit of the current byte of message M(X).
- R_i is the ith bit of the CRC register.
- C_i is the ith 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 entries 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 contents 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)	$X^{16} + X^{12} + X^5 + 1$
CRC-12	$X^{12} + X^{11} + X^3 + X^2 + X + 1$
CRC-16 REVERSE	$X^{16} + X^{14} + X + 1$
SDLC REVERSE	$X^{16} + X^{11} + X^4 + 1$
LRCC-16	$X^{16} + 1$
LRCC-8	$X^8 + 1$

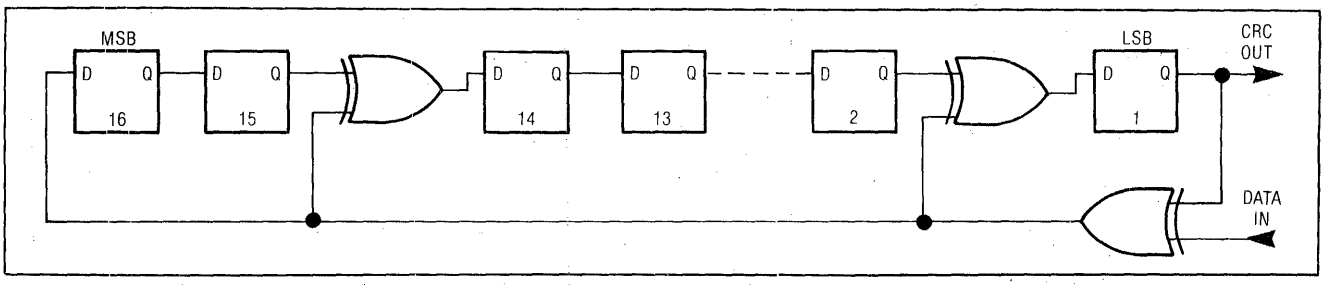


Figure 1. Hardware for CRC-16 calculation.

The exclusive-OR function has the following properties, given here without proofs (⊕ means exclusive-OR):

- A ⊕ B equals B ⊕ A (commutativity).
- A ⊕ B ⊕ C equals A ⊕ C ⊕ B (associativity).
- A ⊕ A equals 0 (involution).
- A ⊕ 0 equals 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 ith bit of the input data byte with the ith 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.

SH	IN	CRC REGISTER															
		R_{16}	R_{15}	R_{14}	R_{13}	R_{12}	R_{11}	R_{10}	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1
0		C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1
1	M_1	C_1 M_1	C_{16}	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2 C_1 M_1
2	M_2	C_2 C_1 M_1 M_2	C_1 M_1	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_8	C_8	C_7	C_6	C_5	C_4	C_3 C_2 C_1 M_1 M_2
3	M_3	C_3 C_2 C_1 M_1 M_2 M_3	C_2 C_1 M_1 M_2	C_1 M_1 C_3 C_2 C_1 M_1 M_2 M_3	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5	C_4 C_3 C_2 C_1 M_1 M_2 M_3
4	M_4	C_4 C_3 C_3 C_1 M_1 M_2 M_3 M_4	C_3 C_2 C_1 M_1 M_2 M_3	C_2 C_1 M_1 M_2 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4	C_1 M_1 C_3 C_2 C_1 M_1 M_2 M_3	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6	C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4
	M_5	C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5	C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4	C_3 C_2 C_1 M_1 M_2 M_3 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5	C_2 C_1 M_1 M_2 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4	C_1 M_1 C_3 C_2 C_1 M_1 M_2 M_3	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7	C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5

Table 2 cont'd

6	M_6	C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6	C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5	C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6	C_3 C_2 C_1 M_1 M_2 M_3 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5	C_2 C_1 M_1 M_2 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4	C_1 M_1 C_3 C_2 C_1 M_1 M_2 M_3	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8	C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6
7	M_7	C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7	C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6	C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7	C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6	C_3 C_2 C_1 M_1 M_2 M_3 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5	C_2 C_1 M_1 M_2 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4	C_1 M_1 C_3 C_2 C_1 M_1 M_2 M_3	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9	C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7
8	M_8	C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8	C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7	C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8	C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7	C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6	C_3 C_2 C_1 M_1 M_2 M_3 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5	C_2 C_1 M_1 M_2 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4	C_1 M_1 C_3 C_2 C_1 M_1 M_2 M_3	C_{16} C_2 C_1 M_1 M_2	C_{15} C_1 M_1	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9 C_8 C_7 C_6 C_5 C_4 C_3 C_2 C_1 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8

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	CRC REGISTER															
		R_{16}	R_{15}	R_{14}	R_{13}	R_{12}	R_{11}	R_{10}	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1
8	M_8	C_8	C_7	C_8	C_7	C_6	C_5	C_4	C_3	C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9
		M_8	M_7	M_8	M_7	M_6	M_5	M_4	M_3	C_2	C_1						C_8
		C_7	C_6	C_7	C_6	C_5	C_4	C_3	C_2	M_2	M_1						M_8
		M_7	M_6	M_7	M_6	M_5	M_4	M_3	M_2	C_1							C_7
		C_6	C_5							M_1							M_7
		M_6	M_5														C_6
		C_5	C_4														M_6
		M_5	M_4														C_5
		C_4	C_3														M_5
		M_4	M_3														C_4
		C_3	C_2														M_4
		M_3	M_2														C_3
		C_2	C_1														M_3
		M_2	M_1														C_2
		C_1															M_2
		M_1															C_1
																	M_1

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

SH	IN	CRC REGISTER															
		R_{16}	R_{15}	R_{14}	R_{13}	R_{12}	R_{11}	R_{10}	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1
8	M_8	X_8	X_7	X_8	X_7	X_6	X_5	X_4	X_3	C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9
		X_7	X_6	X_7	X_6	X_5	X_4	X_3	X_2	X_2	X_1						X_8
		X_6	X_5							X_1							X_7
		X_5	X_4														X_6
		X_4	X_3														X_5
		X_3	X_2														X_4
		X_2	X_1														X_3
		X_1															X_2
																	X_1

Table 5.
Final CRC register.

SH	IN	CRC REGISTER															
		R_{16}	R_{15}	R_{14}	R_{13}	R_{12}	R_{11}	R_{10}	R_9	R_8	R_7	R_6	R_5	R_4	R_3	R_2	R_1
8	M_8	0	0	0	0	0	0	0	0	C_{16}	C_{15}	C_{14}	C_{13}	C_{12}	C_{11}	C_{10}	C_9
		X_8	X_7	X_8	X_7	X_6	X_5	X_4	X_3	X_2	X_1	0	0	0	0	0	X_8
		X_7	X_6	X_7	X_6	X_5	X_4	X_3	X_2	X_1							X_7
		X_6	X_5														X_6
		X_5	X_4														X_5
		X_4	X_3														X_4
		X_3	X_2														X_3
		X_2	X_1														X_2
		X_1															X_1

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 routine 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 6.
Values for byte-wise CRC calculations for all possible X's (values are in hex).

X VALUE	X VALUE	X VALUE	X VALUE	X VALUE	X VALUE	X VALUE	X VALUE
0 0000	32 D801	64 F001	96 2800	128 A001	160 7800	192 5000	224 8801
1 C0C1	33 18C0	65 30C0	97 E8C1	129 60C0	161 B8C1	193 90C1	225 48C0
2 C181	34 1980	66 3180	98 E981	130 6180	162 B981	194 9181	226 4980
3 0140	35 D941	67 F141	99 2940	131 A141	163 7940	195 5140	227 8941
4 C301	36 1B00	68 3300	100 EB01	132 6300	164 BB01	196 9301	228 4B00
5 03C0	37 DBC1	69 F3C1	101 2BC0	133 A3C1	165 7BC0	197 53C0	229 8BC1
6 0280	38 DA81	70 F281	102 2A80	134 A281	166 7A80	198 5280	230 8A81
7 C241	39 1A40	71 3240	103 EA41	135 6240	167 BA41	199 9241	231 4A40
8 C601	40 1E00	72 3600	104 EE01	136 6600	168 BE01	200 9601	232 4E00
9 06C0	41 DEC1	73 F6C1	105 2EC0	137 A6C1	169 7EC0	201 56C0	233 8EC1
10 0780	42 DF81	74 F781	106 2F80	138 A781	170 7F80	202 5780	234 8F81
11 C741	43 1F40	75 3740	107 EF41	139 6740	171 BF41	203 9741	235 4F40
12 0500	44 DD01	76 F501	108 2D00	140 A501	172 7D00	204 5500	236 8D01
13 C5C1	45 1DC0	77 35C0	109 EDC1	141 65C0	173 BDC1	205 95C1	237 4DC0
14 C481	46 1C80	78 3480	110 EC81	142 6480	174 BC81	206 9481	238 4C80
15 0440	47 DC41	79 F441	111 2C40	143 A441	175 7C40	207 5440	239 8C41
16 CC01	48 1400	80 3C00	112 E401	144 6C00	176 B401	208 9C01	240 4400
17 0CC0	49 D4C1	81 FCC1	113 24C0	145 ACC1	177 74C0	209 5CC0	241 84C1
18 0D80	50 D581	82 FD81	114 2580	146 AD81	178 7580	210 5D80	242 8581
19 CD41	51 1540	83 3D40	115 E541	147 6D40	179 B541	211 9D41	243 4540
20 0F00	52 D701	84 FF01	116 2700	148 AF01	180 7700	212 5F00	244 8701
21 CFC1	53 17C0	85 3FC0	117 E7C1	149 6FC0	181 B7C1	213 9FC1	245 47C0
22 CE81	54 1680	86 3E80	118 E681	150 6E80	182 B681	214 9E81	246 4680
23 0E40	55 D641	87 FE41	119 2640	151 AE41	183 7640	215 5E40	247 8641
24 0A00	56 D201	88 FA01	120 2200	152 AA01	184 7200	216 5A00	248 8201
25 CAC1	57 12C0	89 3AC0	121 E2C1	153 6AC0	185 B2C1	217 9AC1	249 42C0
26 CB81	58 1380	90 3B80	122 E381	154 6B80	186 B381	218 9B81	250 4380
27 0B40	59 D341	91 FB41	123 2340	155 AB41	187 7340	219 5B40	251 8341
28 C901	60 1100	92 3900	124 E101	156 6900	188 B101	220 9901	252 4100
29 09C0	61 D1C1	93 F9C1	125 21C0	157 A9C1	189 71C0	221 59C0	253 81C1
30 0880	62 D081	94 F881	126 2080	158 A881	190 7080	222 5880	254 8081
31 C841	63 1040	95 3840	127 E041	159 6840	191 B041	223 9841	255 4040

Table 7.
Comparison of CRC routines.

	NUMBER OF CPU CYCLES TO CALCULATE CRC-16		
BYTES IN MESSAGE	CRCT	CRCF	CRCB
1	161	227	739
8	1169	1669	5668
16	2321	3319	11269
ROUTINE	NUMBER OF BYTES OF MEMORY FOR ROUTINE		
CRCT	540 (INCLUDING TABLE)		
CRCF	43		
CRCB	44		

wise CRC routine, we found that it is up to five times faster but that it takes 12 times as much memory. Concerned with the speed/memory trade-off, we asked, "Since X_8 through X_1 are known, is it possible to calculate the needed value 'on the fly' instead of looking it up in a table?"

The answer turned out to be yes. Listing 3 (program "CRCF") is the result of trying to calculate the needed value "on the fly." It is actually an implementation of the original version of the algorithm. Our objective in developing this program was to produce the needed value (everything below the dotted line in Table 5) in register pair HL. We used several facts which may not be readily apparent from the program listing. First, by looking at everything below the dotted line in Table 5, we can see that

- R_{16} is the same as R_1,
- R_{16} is $X_8 \oplus R_{15}$, and
- R_{14} through R_7 is $X \oplus (X < 1)$, where " < " is shift left.

Second, by letting XX_7 equal $X_7 \oplus X_6 \oplus X_5 \oplus X_4 \oplus X_3 \oplus X_2 \oplus X_1$ and letting XX_8 equal $X_8 \oplus XX_7$, we can show that

- if X_8 equals 0, then XX_8 equals XX_7, and that
- if X_8 equals 1, then XX_8 equals the complement of XX_7.

And third, 8080/8085 processors have a parity flag which is the result of exclusive-ORing all the bits in the A register.

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 CRCT does need almost twelve times as much memory as CRCB. The surprise turned out to be CRCF. Needing one byte less of memory than CRCB, it is nearly four times faster than that routine. ■

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.

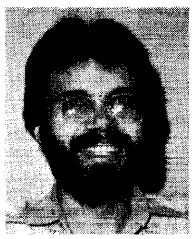
Reference

1. R. Lee, "Cyclic Code Redundancy," *Digital Design*, July 1981, pp. 77-85.

For further reading

J. Martin, *Teleprocessing Network Organization*, Prentice-Hall, Englewood Cliffs, NJ, 1970.

A. K. Pandeya and T. J. Cassa, "Parallel CRC Lets Many Lines Use One Circuit," *Computer Design*, Sept. 1975.



Aram Perez is working on an MSEE at California State University, Sacramento. At the time of the writing of this article, he was a software engineer at Wismer & Becker, also in Sacramento. While with that firm, he was in charge of designing and writing software for microprocessor-controlled data communications systems. Before joining Wismer & Becker, he was a software engineer for Teletek Enterprises, Inc. His interests include home computing, computer graphics, and music. Perez received a BSE from Walla Walla College, College Place, Washington, in 1978. His address is Jones Futura Foundation, 9700 Fair Oaks Blvd., Suite G, Fair Oaks, CA 95628.

Listing 1. Table generator program.

```
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
      IOUT= 6
C
      WRITE(IOUT,'(" X   Value")')
      WRITE(IOUT,'(" --  -----")')
C
      IXH= 1
      IXL= 1
C
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)
```

```
                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
        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
```

```
CRCTO   EQU     $               ;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=0
        RET                 ;END
;
TCRC16  EQU     $               ;CRC TABLE
        INCLUDE &CRCTB          ;(TABLE PRODUCED BY FORTRAN PROGRAM)
;
        END
```

Listing 3. "On the fly" CRC routine.

```
;-----------------------------------------------
; 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= 0
;                 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.XOR.(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
        LXI     D,0             ;INITIALIZE CRC
CRCF0   EQU     $               ;REPEAT
        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     $
        JNC     CRCF2
        XRI     010B            ;    STEP 10
CRCF2   EQU     $
        MOV     H,A             ;    STEP 11
        RAR                     ;    STEP 12
        DAD     H               ;    STEP 13
        DAD     H
        DAD     H
        DAD     H
        DAD     H
        DAD     H
        ORA     L               ;    STEP 14
        XRA     D               ;    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
```