Keywords: 1-Wire cyclic redundancy check (CRC), iButton CRC, ROM ID

APPLICATION NOTE 27

Understanding and Using Cyclic Redundancy Checks with Maxim 1-Wire and iButton Products

Mar 29, 2001

Abstract: All 1-Wire® devices, including iButton® devices, contain an 8-byte unique registration number in read-only memory (ROM). This registration number is used as a unique network address on a 1-Wire bus. To ensure data communication integrity, one byte of each registration number is a 1-Wire CRC byte. This application note explains how to calculate this 8-bit 1-Wire CRC. It also goes on to explain the 16-bit CRC that is used to verify records saved in the memory of the devices. Both the 1-Wire CRC and the CRC-16 are generated in hardware of select 1-Wire devices to validate data.

Introduction

The Maxim iButton products are a family of devices that all communicate over a single wire following a specific command sequence referred to as the 1-Wire Protocol. A key feature of each device is a unique 8-byte ROM code written into each part at the time of manufacture. The components of this 8-byte code can be seen in **Figure 1**. The least significant byte contains a family code that identifies the type of iButton product. For example, the DS1990A has a family code of 01 hex and the DS1922L has a family code of 41 hex. Since multiple devices of the same or different family types can reside on the same 1-Wire bus simultaneously, it is important for the host to determine how to properly access each of the devices that it locates on the 1-Wire bus. The family code provides this information. The next 6 bytes contain a unique serial number that allows multiple devices within the same family code to be distinguished from each other. This unique serial number can be thought of as an "address" for each device on the 1-Wire bus. The entire collection of devices, plus the host, form a type of miniature local area network, or MicroLAN; they all communicate over the single common wire. The most significant byte in the ROM code of each device contains a cyclic redundancy check (CRC) value based on the previous 7 bytes of data for that part. When the host system begins communication with a device, the 8-byte ROM is read, LSB first. If the CRC that is calculated by the host agrees with the CRC contained in byte 7 of ROM data, the communication can be considered valid. If this is not the case, an error has occurred and the ROM code should be read again.

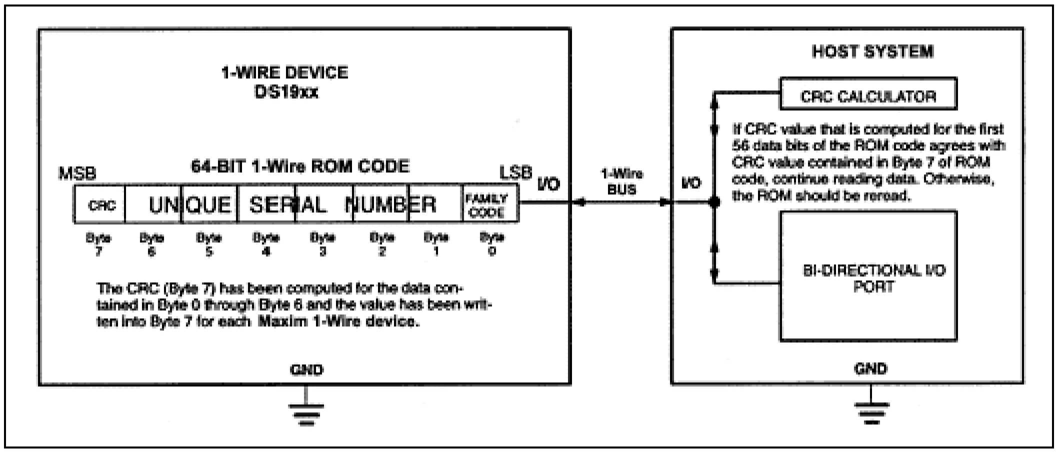

1-WIRE DEVICE
DS19xx
64-BIT 1-Wire ROM CODE
MSB
LSB
CRC
UNIQUE SERIAL NUMBER
FAMILY CODE
Byte 7
Byte 6
Byte 5
Byte 4
Byte 3
Byte 2
Byte 1
Byte 0
The CRC (Byte 7) has been computed for the data contained in Byte 0 through Byte 6 and the value has been written into Byte 7 for each Maxim 1-Wire device.
GND
I/O
1-Wire BUS
I/O
HOST SYSTEM
CRC CALCULATOR
If CRC value that is computed for the first 56 data bits of the ROM code agrees with CRC value contained in Byte 7 of ROM code, continue reading data. Otherwise, the ROM should be reread.
BI-DIRECTIONAL I/O PORT
GND

Figure 1. iButton system configuration using 1-Wire CRC.

Some of the iButton products have up to 8kB of random-access memory (RAM) in addition to the 8 bytes of ROM that can be accessed by the host system with appropriate commands. Even if iButton devices do not have CRC hardware onboard, if the host has the capability to calculate a CRC value for the ROM codes, then a procedure to store and retrieve data in the RAM portion of the devices using CRCs can also be developed. Data can be written to the device in the normal manner; then a CRC value that has been calculated by the host is appended and stored with the data. When this data is retrieved from the iButton device, the process is reversed. The host compares the CRC value that was computed for the data bytes to the value stored in memory as the CRC for that data. If the values are equal, the data read from the iButton device can be considered valid. To take advantage of the power of CRCs to validate the serial communication on the 1-Wire bus, an understanding of what a CRC is and how they work is necessary. In addition, a practical method for calculation of the CRC values by the host are required for either a hardware or software implementation.

Background

Serial data can be checked for errors in a variety of ways. One common way is to include an additional bit in each packet being checked to indicate if an error has occurred. For packets of 8-bit ASCII characters, for example, an extra bit is appended to each ASCII character that indicates if the character contains errors. Suppose the data consisted of a bit string of 11010001. A 9th bit would be appended so the total number of bits that are 1s is always an odd number. Thus, a 1 would be appended and the data packet would become 111010001. The underlined character indicates the parity bit value required to make the complete 9-bit packet have an odd number of bits. If the received data was 111010001, then it would be assumed that the information was correct. If, however, the data received was 111010101, where the 7th bit from the left has been incorrectly received, the total number of 1s is no longer odd and an error condition has been detected and appropriate action would be taken. This type of scheme is called odd parity. Similarly, the total number of 1s could also be chosen to always be equal to an even number, thus the term even parity. This scheme is limited to detecting an odd number of bit errors, however. In the example above, if the data were corrupted and became 111011101, where both the 6th and 7th bits from the left were wrong, the parity check appears correct; yet the error would go undetected whether even or odd parity was used.

Description

Maxim 1-Wire CRC

The error detection scheme most effective at locating errors in a serial-data stream with a minimal amount of hardware is the CRC. The operation and properties of the CRC function used in Maxim products is presented without going into the mathematical details of proving the statements and descriptions. The mathematical concepts behind the properties of the CRC are described in detail in the references. The CRC can be most easily understood by considering the function as it would actually be built in hardware, usually represented as a shift register arrangement with feedback as shown in **Figure 2**. Alternatively, the CRC is sometimes referred to as a polynomial expression in a dummy variable X, with binary coefficients for each of the terms. The coefficients correspond directly to the feedback paths shown in the shift register implementation. The number of stages in the shift register for the hardware description, or the highest order coefficient present in the polynomial expression, indicate the magnitude of the CRC value that is computed. CRC codes that are commonly used in digital data communications include the CRC-16 and the CRC-CCITT, each of which computes a 16-bit CRC value. The Maxim 1-Wire CRC magnitude is 8 bits, which is used for checking the 64-bit ROM code written into each 1-Wire product. This ROM code consists of an 8-bit family code written into the least significant byte, a unique 48-bit serial number written into the next 6 bytes, and a CRC value that is computed based on the preceding 56 bits of ROM and then written into the most significant byte. The location of the feedback paths represented by the exclusive-OR gates in Figure 2, or the presence of coefficients in the polynomial expression, determine the properties of the CRC and the ability of the algorithm to locate certain types of errors in the data. For the 1-Wire CRC, the types of errors that are detectable are:

1. Any odd number of errors anywhere within the 64-bit number.
2. All double-bit errors anywhere within the 64-bit number.
3. Any cluster of errors that can be contained within an 8-bit "window" (1-8 bits incorrect).
4. Most larger clusters of errors.

The input data is exclusive-OR'ed with the output of the eighth stage of the shift register in Figure 2. The shift register can be considered mathematically as a dividing circuit. The input data is the dividend, and the shift register with feedback acts as a divisor. The resulting quotient is discarded, and the remainder is the CRC value for that particular stream of input data, which resides in the shift register after the last data bit has been shifted in. From the shift register implementation it is obvious that the final result (CRC value) is dependent, in a very complex way, on the past history of the bits presented. Therefore, it would take an extremely rare combination of errors to escape detection by this method.

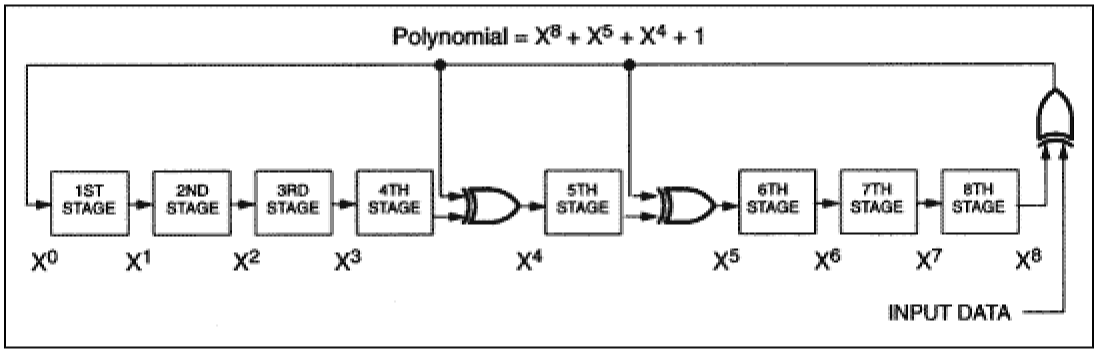


Figure 2. Maxim 1-Wire 8-bit CRC.

The example in Example 2 calculates the CRC value after each data bit is presented. The shift register circuit is always reset to 0s at the start of the calculation. The computation begins with the LSB of the

64-bit ROM, which is the 02 hex family code in this example. After all 56 data bits (serial number + family code) are input, the value that is contained in the shift register is A2 hex, which is the 1-Wire CRC value for that input stream. If the CRC value that has been calculated (A2 hex in this example) is now used as input to the shift register for the next 8 bits of data, the final result in the shift register after the entire 64 bits of data have been entered should be all 0s. This property is always true for the 1-Wire CRC algorithm. If any 8-bit value that appears in the shift register is also used as the next 8 bits in the input stream, then the result that appears in the shift register after the 8th data bit has been shifted in is always 00 hex. This can be explained by observing that the contents of the 8th stage of the shift register is always equal to the incoming data bit, making the output of the EXOR gate controlling the feedback and the next state value of the first stage of the shift register always equal to a logic 0. This causes the shift register to simply shift in 0s from left to right as each data bit is presented, until the entire register is filled with 0s after the 8th bit. The structure of the Maxim 1-Wire 64-bit ROM uses this property to simplify the hardware design of a device used to read the ROM. The shift register in the host is cleared and then the 64 ROM bits are read, including the CRC value. If a correct read has occurred, the shift register is again all 0s, which is an easy condition to detect. If a non-zero value remains in the shift register, the read operation must be repeated.

Until now, the discussion has centered around a hardware representation of the CRC process, but clearly a software solution that parallels the hardware methodology is another means of computing the 1-Wire CRC values. An example of how to code the procedure is given in Example 1. Notice that the XRL (exclusive OR) of the A register with the constant 18 hex is due to the presence of the EXOR feedback gates in the 1-Wire CRC after the fourth and fifth stages as shown in Figure 2. An alternative software solution is to simply build a lookup table that is accessed directly for any 8-bit value currently stored in the CRC register and any 8-bit pattern of new data. For the simple case where the current value of the CRC register is 00 hex, the 256 different bit combinations for the input byte can be evaluated and stored in a matrix, where the index to the matrix is equal to the value of the input byte (i.e., the index is I = 0 to 255). It can be shown that if the current value of the CRC register is not 00 hex, then for any current CRC value and any input byte, the lookup table values are the same as for the simplified case, but the computation of the index into the table takes the form of:

New CRC = Table [I] for I = 0 to 255;
where I = (Current CRC) EXOR (Input byte)

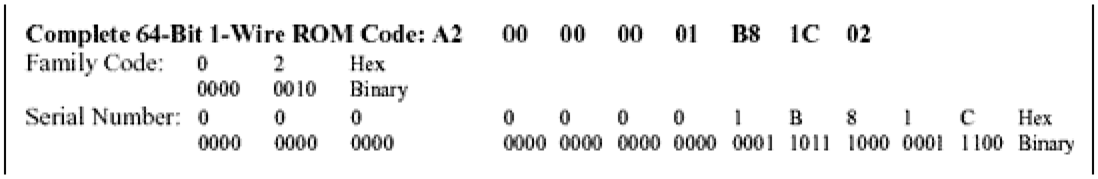
For the case where the current CRC register value is 00 hex, the equation reduces to the simple case. This second approach can reduce computation time since the operation can be done on a byte basis, rather than the bit-oriented commands of the previous example. There is a memory capacity tradeoff, however, since the lookup table must be stored and consumes 256 bytes compared to virtually no storage for the first example except for the program code. An example of this type of code is shown in Example 3. Table 1 shows the previous example repeated using the lookup table approach. Two properties of the 1-Wire CRC can be helpful in debugging code used to calculate the CRC values. The first property has already been mentioned for the hardware implementation. If the current value of the CRC register is used as the next byte of data, the resulting CRC value is always 00 hex (see explanation above). A second property that can be used to confirm proper operation of the code is to enter the 1's complement of the current value of the CRC register. For the 1-Wire CRC algorithm, the resulting CRC value is always 35 hex, or 53 decimal. The reason for this can be explained by observing the operation of the CRC register as the 1's complement data is entered, as shown in **Table 2.**

Example 1. Assembly Language Procedure

```
DO_CRC: PUSH ACC        ;save accumulator
        PUSH B  ;save the B register
        PUSH ACC        ;save bits to be shifted
        MOV B,#8        ;set shift = 8 bits ;

CRC_LOOP:       XRL A,CRC       ;calculate CRC
        RRC A   ;move it to the carry
        MOV A,CRC       ;get the last CRC value
        JNC ZERO        ;skip if data = 0
        XRL A,#18H      ;update the CRC value
;
ZERO:   RRC A   ;position the new CRC
        MOV CRC,A       ;store the new CRC
        POP ACC ;get the remaining bits
        RR A    ;position the next bit
        PUSH ACC        ;save the remaining bits
        DJNZ B,CRC_LOOP ;repeat for eight bits
        POP ACC ;clean up the stack
        POP B   ;restore the B register
        POP ACC ;restore the accumulator
        RET
```

Example 2. Example Calculation for 1-Wire CRC



Complete 64-Bit 1-Wire ROM Code: A2 00 00 00 01 B8 1C 02

Family Code:	0	2	Hex										
	0000	0010	Binary										
Serial Number:	0	0	0	0	0	0	0	1	B	8	1	C	Hex
	0000	0000	0000	0000	0000	0000	0000	0001	1011	1000	0001	1100	Binary

CRC Value	Input Value
00000000	0
00000000	1
10001100	0 2
01000110	0____
00100011	0
10011101	0
11000010	0 0
01100001	0____
10111100	0
01011110	0
00101111	1 C

00010111	1____	
00001011	1	
00000101	0	
10001110	0	1
01000111	0____	
10101111	0	
11011011	0	
11100001	0	8
11111100	1____	
11110010	1	
11110101	1	
01111010	0	B
00111101	1____	
00011110	1	
10000011	0	
11001101	0	1
11101010	0____	
01110101	0	
10110110	0	
01011011	0	0
10100001	0____	
11011100	0	
01101110	0	
00110111	0	0
10010111	0____	
11000111	0	
11101111	0	
11111011	0	0
11110001	0____	
11110100	0	
01111010	0	

00111101	0	0
10010010	0____	
01001001	0	
10101000	0	
01010100	0	0
00101010	0____	
00010101	0	
10000110	0	
01000111	0	0
10101101	0____	
11011010	0	
01101101	0	
10111010	0	0
01011101	0____	
10100010 = A2 hex = CRC Value for [00000001B81C (Serial Number) + 02 (Family Code)]		
10100010	0	
01010001	1	
00101000	0	2
00010100	0____	
00001010	0	
00000101	1	
00000010	0	A
00000001	1____	
00000000 = 00 hex = CRC Value for A2 [(CRC) + 00000001B81C (Serial Number) + 02 (Family Code)]		

Example 3. 1-Wire CRC Lookup Function

```
Var
        CRC : Byte;
Procedure Do_CRC(X: Byte);
{
        This procedure calculates the cumulative Maxim 1-Wire CRC of all
bytes passed to it.
The result accumulates in the global variable CRC.
}
Const
        Table : Array[0..255] of Byte = (

        0, 94, 188, 226, 97, 63, 221, 131, 194, 156, 126, 32, 163, 253, 31,
65,
        157, 195, 33, 127, 252, 162, 64, 30, 95, 1, 227, 189, 62, 96, 130,
220,
        35, 125, 159, 193, 66, 28, 254, 160, 225, 191, 93, 3, 128, 222, 60,
98,
        190, 224, 2, 92, 223, 129, 99, 61, 124, 34, 192, 158, 29, 67, 161,
255,
        70, 24, 250, 164, 39, 121, 155, 197, 132, 218, 56, 102, 229, 187, 89,
7,
        219, 133, 103, 57, 186, 228, 6, 88, 25, 71, 165, 251, 120, 38, 196,
154,
        101, 59, 217, 135, 4, 90, 184, 230, 167, 249, 27, 69, 198, 152, 122,
36,
        248, 166, 68, 26, 153, 199, 37, 123, 58, 100, 134, 216, 91, 5, 231,
185,
        140, 210, 48, 110, 237, 179, 81, 15, 78, 16, 242, 172, 47, 113, 147,
205,
        17, 79, 173, 243, 112, 46, 204, 146, 211, 141, 111, 49, 178, 236, 14,
80,
        175, 241, 19, 77, 206, 144, 114, 44, 109, 51, 209, 143, 12, 82, 176,
238,
        50, 108, 142, 208, 83, 13, 239, 177, 240, 174, 76, 18, 145, 207, 45,
115,
        202, 148, 118, 40, 171, 245, 23, 73, 8, 86, 180, 234, 105, 55, 213,
139,
        87, 9, 235, 181, 54, 104, 138, 212, 149, 203, 41, 119, 244, 170, 72,
22,
        233, 183, 85, 11, 136, 214, 52, 106, 43, 117, 151, 201, 74, 20, 246,
168,
        116, 42, 200, 150, 21, 75, 169, 247, 182, 232, 10, 84, 215, 137, 107,
53);

Begin
        CRC := Table[CRC xor X];
End;
```

Table 1. Table Lookup Method for Computing 1-Wire CRC

Current CRC Value (= Current Table Index)	Input Data	New Index (= Current CRC xor Input Data)	Table (New Index) (= New CRC Value)
0000 0000 = 00 hex	0000 0010 = 02 hex	(00 H xor 02 H) = 02 hex = 2 dec	Table[2]= 1011 1100 = BC hex = 188 dec
	0001		

1011 1100 = BC hex	1100 = 1C hex	(BC H xor 1C H) = A0 hex = 160 dec	Table[160]= 1010 1111 = AF hex = 175 dec
1010 1111 = AF hex	1011 1000 = B8 hex	(AF H xor B8 H) = 17 hex = 23 dec	Table[23]= 0001 1110 = 1E hex = 30 dec
0001 1110 = 1E hex	0000 0001 = 01 hex	(1E H xor 01 H) = 1 F hex = 31 dec	Table[31]= 1101 110 = DC hex = 220 dec
1101 1100 = DC hex	0000 0000 = 00 hex	(DC H xor 00 H) = DC hex = 220 dec	Table[220]= 1111 0100 = F4 hex = 244 dec
11110100 = F4 hex	0000 0000 = 00 hex	(F4 H xor 00 H) = F4 hex = 244 dec	Table [244]= 0001 0101 = 15 hex = 21 dec
0001 0101 = 15 hex	0000 0000 = 00 hex	(15 H xor 00 H) = 15 hex = 21 dec	Table[21]= 1010 0010 = A2 hex = 162 dec
1010 0010 = A2 hex	10100010 = A2 hex	(A2 H xor A2 H) = hex = 0 dec	Table[0]=0000 0000 = 00 hex = 0 dec

CRC Register Combined with 1's Complement of CRC Register

Table 2. CRC Register Value Input

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_7^*
1	X_0	X_1	X_2	X_3^*	X_4^*	X_5	X_6	X_6^*
1	1	X_0	X_1	X_2^*	X_3	X_4^*	X_5	X_5^*
1	1	1	X_0	X_1^*	X_2^*	X_3	X_4^*	X_4^*
0	1	1	1	X_0	X_1^*	X_2	X_3	X_3^*
1	0	1	1	0	X_0^*	X_1^*	X_2	X_2^*
1	1	0	1	0	1	X_0^*	X_1^*	X_1^*
0	1	1	0	1	0	1	X_0^*	X_0^*
0	0	1	1	0	1	0	1	Final CRC Value = 35 hex, 53 decimal

Note: X_i^* = Complement of X_i

CRC-16 Computation for RAM Records in iButton Devices

As mentioned in the introduction, some iButton devices have RAM in addition to the unique 8-byte ROM code found in all iButton devices. Because the amount of data stored in RAM can be large compared to the 8-byte ROM code, Maxim recommends using a 16-bit CRC value to ensure the integrity of the data, rather than the 8-bit 1-Wire CRC used for the ROM. The particular CRC suggested is commonly referred to as CRC-16. The shift register and polynomial representations are given in **Figure 3**. The figure shows that for a 16-bit CRC, the shift register contains 16 stages and the polynomial expression has a term of the sixteenth order. As stated previously, the iButton devices do not calculate the CRC values. The host must generate the value and then append the 16-bit CRC value to the end of the actual data. Due to the uncertainty of the iButton device's "communication channel," i.e., the two metal contact surfaces, data transfers can experience errors that generally fall into three categories. First, brief intermittent connections can cause small numbers of bit errors to occur in the data, which the normal CRC-16 function is designed to detect. The second type of error occurs when contact is lost altogether, for example when the iButton device is removed from the reader too quickly.

This causes the last portion of the data to be read as logic 1s, since no connection to an iButton device is interpreted as all 1s by the host. The normal CRC-16 function can also detect this condition under most circumstances. The third type of error is generated by a short circuit across the reader, which can be caused by an iButton device that is not inserted correctly, or tilted significantly once in the reader. A short at the reader causes the data to be read as all 0s by the host. When using CRCs, this can cause problems, since the method to determine the validity of the data is to read the data plus the stored CRC value, and see if the resulting CRC computed at the host is 0000 hex (for a 16-bit CRC). If the reader was shorted, the data plus the CRC value stored with the data is read as all 0's, and a false read has occurred, but the CRC computed by the host incorrectly indicates a valid read. To avoid this situation, Maxim recommends storing the complement of the computed CRC-16 value (CRC-16*) with the data that is written into the RAM. Using an uncomplemented CRC-16 value, the retrieval of data from the iButton device is similar to the 1-Wire CRC case. That is, if the CRC register in the host is initialized to 0000 hex and then all of the data plus the CRC-16 value stored with the data is read from the iButton device, the resulting calculation by the host should have a 0000 hex, as a final result. If instead, the complement of the CRC-16 value is stored with the data in the iButton, then the CRC register at the host is initialized to 0000 hex and the actual data plus the stored CRC-16* value is read. The resultant CRC value should be B001 hex for a valid read. This greatly improves the operation of the system, since it can no longer be fooled by a short at the reader. The reason that the CRC-16 function has these properties can be shown in an analogous manner to the 1-Wire CRC case (see Figures 3 and 5). The operation of the 16-bit CRC is identical in theory to the 8-bit version described earlier, but the properties of the CRC change since a 16-bit value is now available for error detection. For the CRC-16 function, the types of errors that are detectable are:

1. Any odd number of errors anywhere within the data record.
2. All double-bit errors anywhere within the data record.
3. Any cluster of errors that can be contained within a 16-bit "window" (1–16 bits incorrect).
4. Most larger clusters of errors.

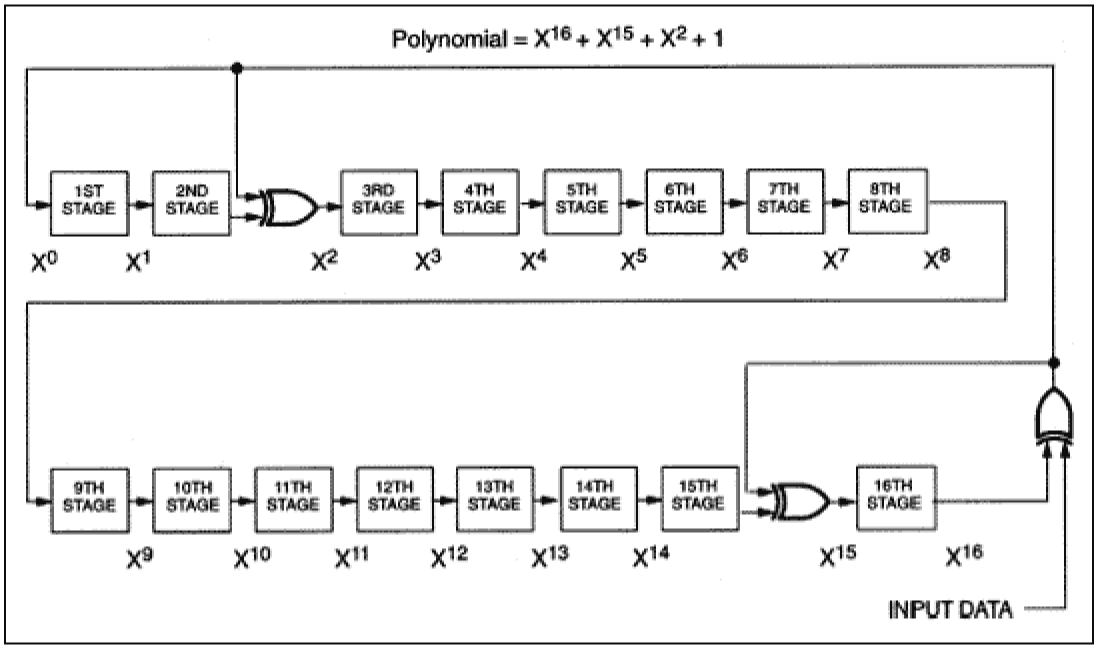


Figure 3. CRC-16 hardware description and polynomial.

The hardware implementation of the CRC-16 function is straightforward from the description given in Figure 3. Example 4 shows a software solution that is analogous to the hardware operations that compute the CRC-16 values using single-bit operations. As before, a less computation-intensive software solution can be developed through the use of a lookup table. The basic concepts presented for the 8-bit 1-Wire CRC lookup table also work for the CRC-16 case. A slight modification in procedure from the 8-bit case is required, however, because if the entire 16-bit result for the CRC-16 function were mapped into one table as before, the table would have 2^{16} or 65,536 entries. A different approach is shown in Example 5, where the 16-bit CRC values are computed and stored in two 256-entry tables, one containing the high order byte and the other the low-order byte of the resultant CRC. For any current 16-bit CRC value, expressed as Current_CRC16_Hi for the current high-order byte and Current_CRC16_Lo for the current low-order byte, and any new input byte, the equation to determine the index into the high-order byte table for locating the new high-order byte CRC value (New_CRC16_Hi) is given as:

New_CRC16_Hi = CRC16_Tabhi[I] for I = 0 to 255; where I = (Current_CRC16_Lo) EXOR (Input byte)

The equation to determine the index into the low-order byte table for locating the new low-order byte CRC value (New_CRC16_Lo) is given as:

New_CRC16_Lo = (CRC16_Tablo[I]) EXOR (Current_CRC16_Hi) for I = 0 to 255;
where I = (Current_CRC16_Lo) EXOR (Input byte)

An example of how this method works is shown in **Figure 4**.

Example 4. Assembly Language for CRC-16 Computation

```
crc_lo data 20h ; lo byte of crc calculation (bit addressable)
crc_hi data 21h ; hi part of crc calculation
```

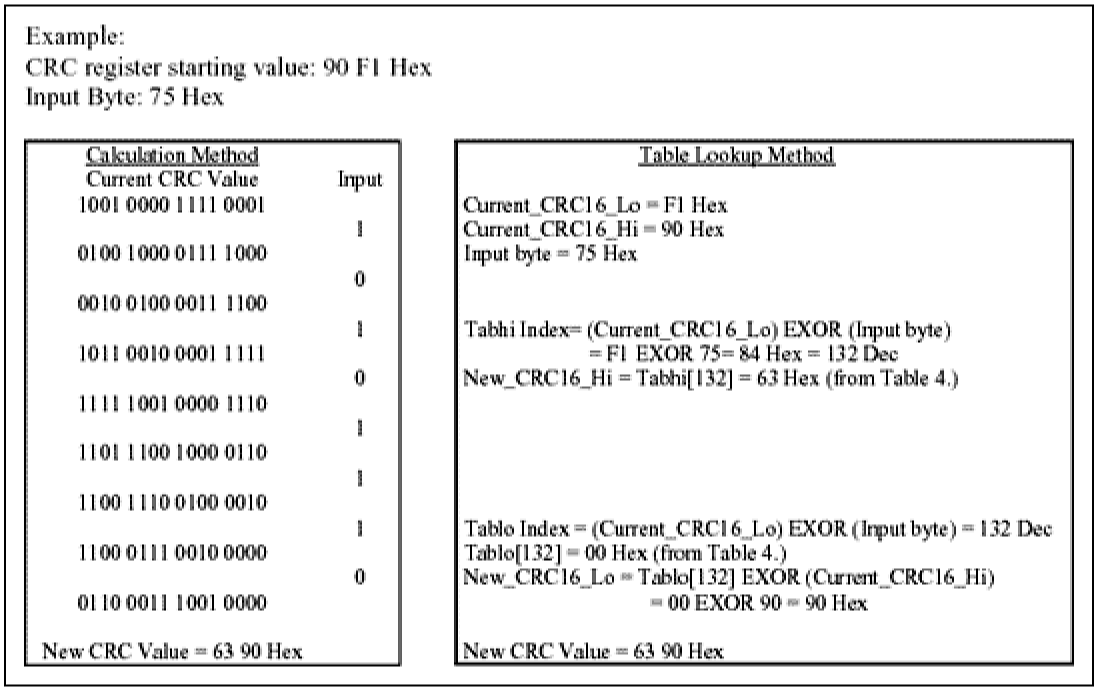
```
;---------------------------------------------------------------------------
;       CRC16 subroutine.
;       - accumulator is assumed to have byte to be crc'ed
;       - two direct variables are used crc_hi and crc_lo
;       - crc_hi and crc_lo contain the CRC16 result
;---------------------------------------------------------------------------
crc16:                  ; calculate crc with accumulator
        push b  ; save value of b
        mov b, #08h     ; number of bits to crc.
crc_get_bit:
        rrc a   ; get low order bit into carry
        push acc        ; save a for later use
        jc crc_in_1     ;got a 1 input to crc
        mov c, crc_lo.0 ;xor with a 0 input bit is bit
        sjmp crc_cont   ;continue
crc_in_1:
        mov c, crc_lo.0 ;xor with a 1 input bit
        cpl c   ;is not bit.
crc_cont:
        jnc crc_shift   ; if carry set, just shift
        cpl crc_hi.6    ;complement bit 15 of crc
        cpl crc_lo.1    ;complement bit 2 of crc
crc_shift
        mov a, crc_hi   ; carry is in appropriate setting
        rrc a ; rotate  it
        mov crc_hi, a   ; and save it
        mov a, crc_lo   ; again, carry is okay
        rrc a ; rotate  it
        mov crc_lo, a   ; and save it
        pop acc ; get acc back
        djnz b, crc_get_bit     ; go get the next bit
        pop b   ; restore b
        ret
        end
```

Example 5. Assembly Language for CRC-16 Using a Lookup Table

```
crc_lo data 40h ; any direct address is okay
crc_hi data 41h
tmp data 42h



;---------------------------------------------------------------------------
;       CRC16 subroutine.
;       - accumulator is assumed to have byte to be crc'ed
;       - three direct variables are used, tmp, crc_hi and crc_lo
;       - crc_hi and crc_lo contain the CRC16 result
;       - this CRC16 algorithm uses a table lookup
;----------------------------------------------------------------------------
crc16:
        xrl a, crc_lo   ; create index into tables
        mov tmp, a      ; save index
        push dph        ; save dptr
        push dpl        ;
        mov dptr, #crc16_tablo  ; low part of table address
        movc a, @a+dptr ; get low byte
        xrl a, crc_hi   ;
        mov crc_lo, a   ; save of low result
        mov dptr, #crc16_tabhi  ; high part of table address
        mov a, tmp      ; index
        movc a, @a+dptr ;
        mov crc_hi, a   ; save high result
        pop dpl ; restore pointer
        pop dph ;
        ret     ; all done with calculation
crc16_tablo:
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 000h, 0c1h, 081h, 040h, 001h, 0c0h, 080h, 041h
        db 001h, 0c0h, 080h, 041h, 000h, 0c1h, 081h, 040h
crc16_tabhi:
        db 000h, 0c0h, 0c1h, 001h, 0c3h, 003h, 002h, 0c2h
        db 0c6h, 006h, 007h, 0c7h, 005h, 0c5h, 0c4h, 004h
        db 0cch, 00ch, 00dh, 0cdh, 00fh, 0cfh, 0ceh, 00eh
        db 00ah, 0cah, 0cbh, 00bh, 0c9h, 009h, 008h, 0c8h
        db 0d8h, 018h, 019h, 0d9h, 01bh, 0dbh, 0dah, 01ah
        db 01eh, 0deh, 0dfh, 01fh, 0ddh, 01dh, 01ch, 0dch
        db 014h, 0d4h, 0d5h, 015h, 0d7h, 017h, 016h, 0d6h
        db 0d2h, 012h, 013h, 0d3h, 011h, 0d1h, 0d0h, 010h
        db 0f0h, 030h, 031h, 0f1h, 033h, 0f3h, 0f2h, 032h
        db 036h, 0f6h, 0f7h, 037h, 0f5h, 035h, 034h, 0f4h
        db 03ch, 0fch, 0fdh, 03dh, 0ffh, 03fh, 03eh, 0feh
        db 0fah, 03ah, 03bh, 0fbh, 039h, 0f9h, 0f8h, 038h
        db 028h, 0e8h, 0e9h, 029h, 0ebh, 02bh, 02ah, 0eah
        db 0eeh, 02eh, 02fh, 0efh, 02dh, 0edh, 0ech, 02ch
        db 0e4h, 024h, 025h, 0e5h, 027h, 0e7h, 0e6h, 026h
        db 022h, 0e2h, 0e3h, 023h, 0e1h, 021h, 020h, 0e0h
        db 0a0h, 060h, 061h, 0a1h, 063h, 0a3h, 0a2h, 062h
        db 066h, 0a6h, 0a7h, 067h, 0a5h, 065h, 064h, 0a4h
        db 06ch, 0ach, 0adh, 06dh, 0afh, 06fh, 06eh, 0aeh
        db 0aah, 06ah, 06bh, 0abh, 069h, 0a9h, 0a8h, 068h
        db 078h, 0b8h, 0b9h, 079h, 0bbh, 07bh, 07ah, 0bah
        db 0beh, 07eh, 07fh, 0bfh, 07dh, 0bdh, 0bch, 07ch
        db 0b4h, 074h, 075h, 0b5h, 077h, 0b7h, 0b6h, 076h
        db 072h, 0b2h, 0b3h, 073h, 0b1h, 071h, 070h, 0b0h
        db 050h, 090h, 091h, 051h, 093h, 053h, 052h, 092h
        db 096h, 056h, 057h, 097h, 055h, 095h, 094h, 054h
        db 09ch, 05ch, 05dh, 09dh, 05fh, 09fh, 09eh, 05eh
        db 05ah, 09ah, 09bh, 05bh, 099h, 059h, 058h, 098h
        db 088h, 048h, 049h, 089h, 04bh, 08bh, 08ah, 04ah
        db 04eh, 08eh, 08fh, 04fh, 08dh, 04dh, 04ch, 08ch
        db 044h, 084h, 085h, 045h, 087h, 047h, 046h, 086h
        db 082h, 042h, 043h, 083h, 041h, 081h, 080h, 040h
```


Example:
CRC register starting value: 90 F1 Hex
Input Byte: 75 Hex

Calculation Method

Current CRC Value	Input
1001 0000 1111 0001	
	1
0100 1000 0111 1000	
	0
0010 0100 0011 1100	
	1
1011 0010 0001 1111	
	0
1111 1001 0000 1110	
	1
1101 1100 1000 0110	
	1
1100 1110 0100 0010	
	1
1100 0111 0010 0000	
	0
0110 0011 1001 0000	

New CRC Value = 63 90 Hex

Table Lookup Method

Current_CRC16_Lo = F1 Hex
Current_CRC16_Hi = 90 Hex
Input byte = 75 Hex

Tabhi Index= (Current_CRC16_Lo) EXOR (Input byte)
= F1 EXOR 75= 84 Hex = 132 Dec
New_CRC16_Hi = Tabhi[132] = 63 Hex (from Table 4.)

Tablo Index = (Current_CRC16_Lo) EXOR (Input byte) = 132 Dec
Tablo[132] = 00 Hex (from Table 4.)
New_CRC16_Lo = Tablo[132] EXOR (Current_CRC16_Hi)
= 00 EXOR 90 = 90 Hex

New CRC Value = 63 90 Hex

Figure 4. Comparison of calculation and table lookup method for CRC-16.

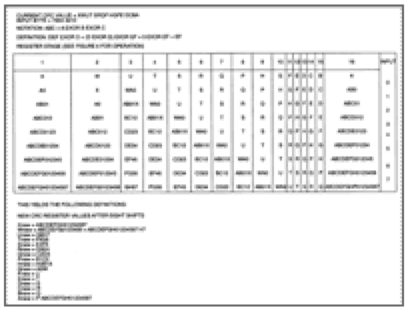
An interesting intermediate method is presented in Example 6. The code generates a CRC-16 value for each byte input to it by operating on the entire current CRC value and the incoming byte using the equations shown in **Figure 5**. The derivations for the equations are also shown, using alpha characters to represent the current 16-bit CRC value and numeric characters to represent the bits of the incoming byte. The result after eight shifts yields the equations shown. These equations can then be used to precompute large portions of the new CRC value. Notice, for example, that the quantity ABCDEFGH01234567 (defined as the EXOR of all of those bits) is the parity of the incoming data byte and the low-order byte of the current CRC. This method reduces computation time and memory space as compared to both the bit-by-bit and lookup table methods described above. Finally, two properties of the CRC-16 function that can be used as test cases are mentioned as an aid to debugging the code for any of the previous methods. The first property is identical to the 1-Wire CRC case. If the current 16-bit contents of the CRC register are also used as the next 16 bits of input, the resulting CRC value is always 0000 hex. A second property of the CRC-16 function is also similar to the 1-Wire CRC case, if the 1's complement of the current 16-bit contents of the CRC register are also used as the next 16-bits of input, the resulting CRC value is always B0 01 hex. The proof for these two CRC-16 properties follows in an analogous way to the proof for the 1-Wire CRC case.

Example 6. Assembly Language Procedure for High-Speed CRC-16 Computation

```
lo equ 40h ; low byte of CRC
hi equ 41h ; high byte of CRC



crc16:
        push acc        ; save the accumulator.
        xrl a, lo
        mov lo, hi      ; move the high byte of the CRC.
        mov hi, a       ; save data xor low(crc) for later
        mov c, p
        jnc crc0
        xrl lo, #01h    ; add the parity to CRC bit 0
crc0:
        rrc a   ; get the low bit in c
        jnc crc1
        xrl lo, #40h    ; need to fix bit 6 of the result
crc1:
        mov c, acc.7
        xrl a, hi       ; compute the results for other bits.
        rrc a   ; shift them into place
        mov hi, a       ; and save them
        jnc crc2
        xrl lo, #80h    ; now clean up bit 7
crc2:
        pop acc ; restore everything and return
        ret
```



More detailed image (GIF)

Figure 5. High-speed CRC-16 computation method.

References

Stallings, William, Ph.D., *Data and Computer Communications.* 2nd ed., New York: Macmillan Publishing. pp. 107–112.
Buller, Jon, "High Speed Software CRC Generation", EDN, Volume 36, #25, p. 210.

Related Parts		
DS1822	Econo 1-Wire Digital Thermometer	Free Samples
DS18B20	Programmable Resolution 1-Wire Digital Thermometer	Free Samples

DS18B20-PAR	1-Wire Parasite-Power Digital Thermometer	
DS18S20	1-Wire Parasite-Power Digital Thermometer	Free Samples
DS18S20-PAR	Parasite-Power Digital Thermometer	
DS1904	iButton RTC	Free Samples
DS1920	iButton Temperature Logger	
DS1921G	Thermochron iButton Device	
DS1963S	iButton Monetary Device with SHA-1 Function	
DS1971	iButton 256-Bit EEPROM	
DS1973	iButton 4Kb EEPROM	Free Samples
DS1982	iButton 1Kb Add-Only	Free Samples
DS1985	iButton 16Kb Add-Only	Free Samples
DS1986	iButton 64Kb Add-Only	
DS1990A	iButton Serial Number	Free Samples
DS1992	iButton 1Kb/4Kb Memory	Free Samples
DS1993	iButton 1Kb/4Kb Memory	Free Samples
DS1995	iButton 16Kb Memory	Free Samples
DS1996	iButton 64Kb Memory	Free Samples
DS2401	Silicon Serial Number	Free Samples
DS2405	Addressable Switch	
DS2406	Dual Addressable Switch Plus 1Kb Memory	Free Samples
DS2408	1-Wire 8-Channel Addressable Switch	Free Samples
DS2411	Silicon Serial Number with V_{CC} Input	Free Samples
DS2415	1-Wire Time Chip	
DS2417	1-Wire Time Chip With Interrupt	Free Samples
DS2431	1024-Bit 1-Wire EEPROM	Free Samples
DS2432	1Kb Protected 1-Wire EEPROM with SHA-1 Engine	Free Samples
DS2433	4Kb 1-Wire EEPROM	
DS2438	Smart Battery Monitor	Free Samples
DS2450	1-Wire Quad A/D Converter	
DS2502	1Kb Add-Only Memory	Free Samples
DS2502-E48	48-Bit Node Address Chip	Free Samples

DS2505	16Kb Add-Only Memory	Free Samples
DS2506	64Kb Add-Only Memory	
DS2760	High-Precision Li+ Battery Monitor	
MAX31820	1-Wire Ambient Temperature Sensor	Free Samples
MAX31820PAR	1-Wire Parasite-Power, Ambient Temperature Sensor	
MAX31826	1-Wire Digital Temperature Sensor with 1Kb Lockable EEPROM	Free Samples