INTRODUCTION

This application note describes the Cyclic Redundancy Check (CRC) theory and implementation. The CRC check is used to detect errors in a message. Two implementations are shown:

- Table driven CRC calculation
- Loop driven CRC calculation

This application describes the implementation of the CRC-16 polynomial. However, there are several formats for the implementation of CRC such as CRC-CCITT, CRC-32 or other polynomials.

CRC is a common method for detecting errors in transmitted messages or stored data. The CRC is a very powerful, but easily implemented technique to obtain data reliability.

THEORY OF OPERATION

The theory of a CRC calculation is straightforward. The data is treated by the CRC algorithm as a binary number. This number is divided by another binary number called the polynomial. The rest of the division is the CRC checksum, which is appended to the transmitted message. The receiver divides the message (including the calculated CRC), by the same polynomial the transmitter used. If the result of this division is zero, then the transmission was successful. However, if the result is not equal to zero, an error occurred during the transmission.

The CRC-16 polynomial is shown in Equation 1. The polynomial can be translated into a binary value, because the divisor is viewed as a polynomial with binary coefficients. For example, the CRC-16 polynomial translates to 1000000000000101b. All coefficients, like \(x^2\) or \(x^{15}\), are represented by a logical 1 in the binary value.

The division uses the Modulo-2 arithmetic. Modulo-2 calculation is simply realized by XOR'ing two numbers.

EXAMPLE 1: MODULO-2 CALCULATION

```
1 0 0 1 1 0 0 1 0 1
XOR 0 1 0 0 1 1 0 1 1 1
   = 1 1 0 1 0 1 0 0 1 0
```

XOR-Function

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

EQUATION 1: THE CRC-16 POLYNOMIAL

\[ P(x) = x^{16} + x^{15} + x^2 + 1 \]

Example Calculation

In this example calculation, the message is two bytes long. In general, the message can have any length in bytes. Before we can start calculating the CRC value 1, the message has to be augmented by n-bits, where n is the length of the polynomial. The CRC-16 polynomial has a length of 16-bits, therefore, 16-bits have to be augmented to the original message. In this example calculation, the polynomial has a length of 3-bits, therefore, the message has to be extended by three zeros at the end. An example calculation for a CRC is shown in Example 1. The reverse calculation is shown in Example 2.
EXAMPLE 2:  CALCULATION FOR GENERATING A CRC

Message = 110101
Polynomial = 101

\[
\begin{array}{c}
11010100 \div 101 = 11101 \\
101 \\
111 \\
101 \\
100 \\
101 \\
110 \\
101 \\
110 \\
101 \\
11 \\
\end{array}
\]

Remainder = CRC checksum

Message with CRC = 11010111

EXAMPLE 3:  CHECKING A MESSAGE FOR A CRC ERROR

Message with CRC = 11010111
Polynomial = 101

\[
\begin{array}{c}
11010111 \div 101 = 11101 \\
101 \\
111 \\
101 \\
100 \\
101 \\
111 \\
101 \\
101 \\
101 \\
101 \\
00 \\
\end{array}
\]

Checksum is zero, therefore, no transmission error
CRC Hardware Implementation

The CRC calculation is realized with a shift register and XOR gates. Figure 1 shows a CRC generator for the CRC-16 polynomial. Each bit of the data is shifted into the CRC shift register (Flip-Flops) after being XOR'ed with the CRC’s most significant bit.

Software Implementations

There are two different techniques for implementing a CRC in software. One is a loop driven implementation and the other is a table driven implementation.

The loop driven implementation works like the calculation shown in Figure 2. The data is fed through a shift register. If a one pops out at the MSb, the content is XORed with the polynomial. In the other, the registers are shifted by one position to the left.

Another method to calculate a CRC is to use precalculated values and XOR them to the shift register.

Note: The mathematical details are not given within this application note. The interested reader may refer to the material shown in the Reference section.
LOOP DRIVEN CRC IMPLEMENTATION

This section describes the loop driven CRC implementation. This implementation is derived from the hardware implementation shown in Figure 1. The program for the loop driven CRC generation and CRC checking is shown in Appendix A.

CRC Generation

The implementation of a loop driven CRC generation is shown in Figure 2. In the first step the registers, CRC_HIGH and CRC_LOW, are initialized with the first two bytes of data. CRC_BUFF is loaded with the third byte of data. After that, the MSb of CRC_BUFF is shifted into the LSb of CRC_LOW and the MSb of CRC_LOW is shifted into the LSb of CRC_HIGH. The MSb of CRC_HIGH, which is now stored in the Carry flag, is tested to see if it is set. If the bit is set, the registers, CRC_HIGH and CRC_LOW, will be XORed with the CRC-16 polynomial. If the bit is not set, the next bit from CRC_BUFF will be shifted into the LSb of CRC_LOW.

This process is repeated until all data from CRC_BUFF is shifted into CRC_LOW. After this, CRC_BUFF is loaded with the next data byte. Then all data bytes are processed, and 16 zeros are appended to the message. The registers, CRC_HIGH and CRC_LOW, contain the calculated CRC value. The message can have any length. In this application note, the CRC value for 8 data bytes is calculated.

CRC Checking

The CRC check uses the same technique as the CRC generation, with the only difference being that zeros are not appended to the message.

TABLE DRIVEN CRC IMPLEMENTATION

A table driven CRC routine uses a different technique than a loop driven CRC routine. The idea behind a table driven CRC implementation is that instead of calculating the CRC bit by bit, precomputed bytes are XORed to the data. The source code for the table driven implementation is given in Appendix B.

The advantage of the table driven implementation is that it is faster than the loop driven solution. The drawback is that it consumes more program memory because of the size of the look-up table.

CRC Generation

The CRC at the table driven implementation is generated by reading a precomputed value out of a table and XOR, the result with the low and high byte of the CRC shift registers.

In the first step, the registers, CRC_BUFF, CRC_HIGH and CRC_LOW, are initialized with the first three bytes of data. After that, the value in CRC_BUFF is used as an offset to get the value for the precomputed CRC value out of the look-up table. Since the CRC-16 is 16 bits long, the look-up table is split up into two separate look-up tables. One for the high byte of the CRC register and one for the low byte of the CRC register (see Figure 3). The result from the look-up table of the high byte is XORed to the content of the CRC_HIGH register. The result for the low byte is XORed to the content of CRC_LOW. The results are stored back in CRC_LOW.

The next step is that the content from CRC_HIGH is shifted into CRC_BUFF and the content from CRC_LOW is shifted into the register, CRC_HIGH. Then the register, CRC_LOW, is loaded with the new data byte.

This process repeats for all data bytes. At the end of the CRC generation, the message has to be appended by 16 zeros. The 16 zeros are treated like the data bytes.

After the calculation is done, the content of the registers, CRC_HIGH and CRC_LOW, are appended to the message.

CRC Checking

The CRC check uses the same technique as the CRC generation, with the difference being that zeros are not appended to the message.

COMPARISON

Table 1 shows a comparison between the loop driven implementation and the table driven implementation. For the calculation, 8 data bytes were used.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CRC Generation (in cycles)</th>
<th>CRC Check (in cycles)</th>
<th>Program Memory Usage (words)</th>
<th>Data Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Driven</td>
<td>865</td>
<td>870</td>
<td>85</td>
<td>6</td>
</tr>
<tr>
<td>Table Driven</td>
<td>348</td>
<td>359</td>
<td>612</td>
<td>5</td>
</tr>
</tbody>
</table>
ADVANTAGES OF CRC VS. SIMPLE SUM TECHNIQUES

The CRC generation has many advantages over simple sum techniques or parity check. CRC error correction allows detection of:

1. single bit errors
2. double bit errors
3. bundled bit errors (bits next to each other)

A parity bit check detects only single bit errors. The CRC error correction is mostly used where large data packages are transmitted, for example, in local area networks such as Ethernet.

References

Ross N. Williams - A Painless Guide to CRC Error Detection Algorithms
Donald E. Knuth - The Art of Computer Programming, Volume 2, Addison Wesley
FIGURE 3: TABLE DRIVEN CRC CALCULATION IMPLEMENTATION
APPENDIX A: SOURCE CODE FOR LOOP DRIVEN CRC IMPLEMENTATION

MPASM 02.30.11 Intermediate CRC16_04.ASM  3-9-2000  13:00:00 PAGE 1

LOC OBJECT CODE LINE SOURCE TEXT
VALUE

00001 ; ******************************************************
00002 ; * Title           : CRC16 calculation*              
00003 ; * Author          : Thomas Schmidt  
00004 ; * Date            : 15.04.1999* 
00005 ; * Revision        : 0.4*  
00006 ; * Last Modified   : 15.04.1999*  
00007 ; * Core            : 12-bit, 14-bit (12-bit core tested)*  
00008 ; * Peripherals used: none* 
00009 ; * Registers used  :*  
00010 ; * Modifications   : crc16_01.asm Checksum check was added*  
00011 ; * crc16_03.asm Number of data bytes was increased from 2 to 8 bytes*  
00012 ; * crc16_04.asm added revers CRC*  
00013 ; * Description     :*  
00014 ; *  
00015 ; * This module calculates the checksum for the CRC16 polynom. The CRC16 polynome is defined*  
00016 ; * as x16+x15+x2+x0. The calculation is done by bitwise checking. The algorithm is designed*  
00017 ; * for a two byte wide message. The algorithm can easily be modified for longer messages. The*  
00018 ; * only thing what has to be done is to check after the low byte is shifted into the high byte*  
00019 ; * that the low byte is loaded with a new data byte. The number of iteration has to be adjusted*  
00020 ; * to the number of extra bytes in the data message. The number is calculated as follows:*  
00021 ; * n=16+x*messagebits. For example if the message is 4 bytes long the number of iterations is*  
00022 ; * n=16+16bits. The extra iterations have to be done because the message is extended with 16*
* zeros at the end of the message. *

LIST P=16C54B, r=hex
#include "p16c5x.inc"

#define PolynomLow b'00000101' ; low byte of polynome
#define PolynomHigh b'10000000' ; high byte of polynome
#define PolynomLength 0x10 ; 16-bit polynome length
#define DataLength 0x09 ; Data length in Bits
#define Iterations 0x08 ; number of iterations for CRC calculation

cblock 0x07

CRC_HIGH ; CRC shift registers
CRC_LOW ; second CRC shift register
CRC_BUFF ; CRC buffer register
BITS ; number of data bits
DATABYTES ; number of bytes of data
TEMP ; temporary register

endc

ORG 0x1ff
goto Begin

ORG 0x0000

Begin movlw 0x10 ; Data for CRC generation
movwf FSR ; Set point to begin of data block
movlw 0xAA ; data

; initialization what has to be done before CRC16 routine can be
called. The FSR register contains the pointer to the first byte of
data and the register DATABYTES contains the number of data bytes
of the message.

; set pointer to first data location
movlw 0x10
movwf FSR ; initialize FSR register

Main call CRC16Generate ; calculate CRC16 value
; append CRC to message
incf FSR, f ; point to position behind last data byte
movf CRC_HIGH, w ; copy CRC_HIGH data into w-register
movwf INDF ; copy CRC_HIGH being last data byte
incf FSR, f ; point to next location
movf CRC_LOW, w ; copy CRC_LOW data into w-register
movwf INDF ; copy data into next location
movlw 0x10 ; set pointer to first data location
movwf FSR ; initialize FSR register
call CRC16Restore ; restore CRC16 value

stop goto Stop ; do forever

; *******************************************************************************
; * Title:              CRC16 calculation                                       *
; * Input parameter:    Pointer to first data byte (pointer in FSR register)   *
; *                     Number of data bytes stored in register DATABYTES       *
; * Output:             CRC result stored in CRC_HIGH and CRC_LOW               *
; *******************************************************************************

call CRC16Init ; initialize registers
movlw 0x03 ; initialize TEMP register with 2
movwf TEMP ; move 0x02 into TEMP

NextCRC16 call CRC16Engine ; Calculate CRC16 for one byte
decfz DATABYTES, f ; Decrement the number of data bytes by one
rclw 0x00 ; reload CRC_BUFF register with new data byte
goto Reload ; move 0x02 into TEMP

decfz TEMP, f ; decrement TEMP register
goto AppendZeros ; Append zeros to message
retlw 0x00 ; return to main

cirf CRC_BUFF ; append data with zeros
goto NextCRC16 ; calculate next CRC

decfz CRC_BUFF, f ; increment data bytes

NextCRC16 ; Reload CRC buffer register with new data word.
incf FSR, f ; point to next data byte
movf INDF, w ; copy data into w-register
movwf CRC_BUFF ; move data into CRC_BUFF register
movlw Iterations ; initialize register BITS register
movwf BITS ; with 0x08
movw DATABYTES, f ; increment data bytes

goto NextCRC16 ; last iteration

; *******************************************************************************

00113 ; * Title: Restore CRC function
00114 ; * Input: Pointer to first data byte in FSR register
00115 ; * Output: w=0 CRC was restore sucessful
00116 ; * w=1 CRC was not restored sucessful
00117 ; *******************************************************************************
00121
0024 0938 00118 CRC16Restore call CRC16Init ; initialize CRC registers
0025 0C02 00119         movlw 0x02         ; add offset to DATABYTES
0026 01EB 00120         addwf DATABYTES, f    ; add offset to register DATABYTES
0027 0947 00121
0028 02EB 00122 NextCRCRestore call CRC16Engine
0029 0A32 00123         decfsz DATABYTES, f  ; Decrement the number of data bytes by one
002A 0227 00124         movf CRC_HIGH, f     ; check if CRC_HIGH and CRC_LOW equal to zero
002B 0743 00125         btfss STATUS, Z       ; is content zero?
002C 0A31 00126         goto CRCError         ; no, CRC error occurred
002D 0228 00127         movf CRC_LOW, f      ; copy CRC_LOW register onto itself
002E 0743 00128         btfss STATUS, Z       ; is content zero?
002F 0A31 00129         goto CRCError         ; no, CRC error occurred
0030 0800 00130         retlw 0x00          ; return to main (0= no error)
0031 0801 00131 CRCError retlw 0x01          ; return to main with error code 1
0032 02A4 00132 ReloadRestore incf FSR, f     ; point to next data byte
0033 0200 00133         movf INDF, w         ; copy data into w-register
0034 0029 00134         movwf CRC_HIGH, w    ; copy CRC_HIGH into CRC_HIGH register
0035 0042 00135         movwf CRC Buff, w    ; move data into CRC_BUFF register
0036 002A 00136         movwf BITS, w        ; initialize register BITS with 8
0037 0A27 00137         goto NextCRCRestore ; calculate next CRC
0038 00145 00138         ; Reload CRC buffer register with new data word.
0039 00146
0040 00147 ; *******************************************************************************
0041 00148 ; * Title: CRC16 Initialization
0042 00149 ; * Input: Pointer to first data byte in FSR register
0043 00150 ; * Output: none
0044 00151 ; *******************************************************************************
0045 00152 CRC16Init movf INDF, w ; copy data into W-register
0046 00153 movwf CRC_HIGH ; copy w-register into CRC_HIGH register
0047 00154 incf FSR, f ; set pointer to next location
0048 00155 movf INDF, w ; copy data into w-register
0049 00156 movwf CRC_LOW ; copy w-register into CRC_LOW
0050 00157 incf FSR, f ; set pointer to next location
0051 00158 movf INDF, w ; copy next data into w-register
movwf CRC_BUFF ; copy data into CRC buffer register
movlw Iterations ; initialize register BITS with length of polynomial for iteration
movwf BITS ; copy number of data bytes into register DataBytes
movlw DataLength ; initialize register DataBytes
movlw 0x03 ; decrement three from the number
subwf DATABYTES, f ; of data bytes, because three register are now initialized
retlw 0x00 ; return from subroutine

; ***************************************************************************
; * Title: CRC16 Engine                                                     *
; * Input: Pointer to first data byte in FSR register                      *
; * Output: none                                                           *
; ***************************************************************************
bcf STATUS, C ; clear carry flag
rlf CRC_BUFF, f ; shift bit7 of CRC_BUFF into carry flag
rlf CRC_LOW, f ; shift bit7 of CRC_LOW into carry flag
; and shift 0 into bit7 of CRC_LOW
rlf CRC_HIGH, f ; rotate carry flag into bit0 of CRC_HIGH
; and rotate bit7 of CRC_HIGH into carry flag
btfss STATUS, C ; is carry flag set?
; goto NextBitEngine ; carry flag is clear there next rotation
goto NextBitEngine ; carry flag is set therefore XOR CRCSHIFT ; registers
movlw PolynomHigh ; load w-register with low byte of polynomial
movlw PolynomHigh ; XOR CRC_HIGH register
movlw PolynomLow ; load w-register with low byte of polynomial
xorwf CRC_LOW, f ; XOR CRC_LOW register
decfsz BITS, f ; do for all bits
goto CRC16Engine ; calculate CRC for next bit
retlw 0x00 ; return from Subroutine

END
APPENDIX B: SOURCE CODE TABLE DRIVEN CRC IMPLEMENTATION

MPASM 02.30.11 Intermediate CRCTAB01.ASM 3-9-2000 13:02:59 PAGE 1

LOC OBJECT CODE LINE SOURCE TEXT

VALUE

00001 ; **************************************************************************
00002 ; * Title           : CRC16 calculation table driven implementation  *
00003 ; * Author          : Thomas Schmidt                            *
00004 ; * Date            : 22.03.1999                               *
00005 ; * Revision        : 0.1                                     *
00006 ; * Last Modified   : 15.04.1999                              *
00007 ; * Core            : 12-bit, 14-bit (12-bit core tested)      *
00008 ; * Peripherals used: none                                   *
00009 ; * Registers used  :                                       *
00010 ; * Modifications  : crctab01.asm: first program CRC generation *
00011 ; * Description     :                                       *
00012 ; *                                                                 *
00013 ; * This module calculates the checksum for the CRC16 polynom. The CRC16 polynome is defined *
00014 ; * as x16+x15+x2+x0. The calculation is done by bitwise checking. The algorithm is designed *
00015 ; * for a two byte wide message. The algorithm can easily be modified for longer messages. The *
00016 ; * only thing what has to be done is to check after the low byte is shifted into the high byte *
00017 ; * that the low byte is loaded with a new data byte. The number of iteration has to be adjusted *
00018 ; * to the number of extra bytes in the data message. The number is calculated as follows: *
00019 ; * n=16+x*messagebits. For example if the message is 4 bytes long the number of iterations is *
00020 ; * n=16+16bits. The extra iterations have to be done because the message is extended with 16 *
00021 ; * zeros at the end of the message.                               *
00022 ; **************************************************************************
00023
00024 LIST P-16C58B, r-hex
00025
00026 #include "p16c5x.inc"
00027
00028 cblock 0x07 00000007 ; low byte of CRC register
00029 cblock 0x07 00000008 ; high byte of CRC register

00000001 ; **************************************************************************
00000002 ; * Title           : CRC16 calculation table driven implementation  *
00000003 ; * Author          : Thomas Schmidt                            *
00000004 ; * Date            : 22.03.1999                               *
00000005 ; * Revision        : 0.1                                     *
00000006 ; * Last Modified   : 15.04.1999                              *
00000007 ; * Core            : 12-bit, 14-bit (12-bit core tested)      *
00000008 ; * Peripherals used: none                                   *
00000009 ; * Registers used  :                                       *
00000010 ; * Modifications  : crctab01.asm: first program CRC generation *
00000011 ; * Description     :                                       *
00000012 ; *                                                                 *
00000013 ; * This module calculates the checksum for the CRC16 polynom. The CRC16 polynome is defined *
00000014 ; * as x16+x15+x2+x0. The calculation is done by bitwise checking. The algorithm is designed *
00000015 ; * for a two byte wide message. The algorithm can easily be modified for longer messages. The *
00000016 ; * only thing what has to be done is to check after the low byte is shifted into the high byte *
00000017 ; * that the low byte is loaded with a new data byte. The number of iteration has to be adjusted *
00000018 ; * to the number of extra bytes in the data message. The number is calculated as follows: *
00000019 ; * n=16+x*messagebits. For example if the message is 4 bytes long the number of iterations is *
00000020 ; * n=16+16bits. The extra iterations have to be done because the message is extended with 16 *
00000021 ; * zeros at the end of the message.                               *

000000001 ; **************************************************************************
000000002 ; * Title           : CRC16 calculation table driven implementation  *
000000003 ; * Author          : Thomas Schmidt                            *
000000004 ; * Date            : 22.03.1999                               *
000000005 ; * Revision        : 0.1                                     *
000000006 ; * Last Modified   : 15.04.1999                              *
000000007 ; * Core            : 12-bit, 14-bit (12-bit core tested)      *
000000008 ; * Peripherals used: none                                   *
000000009 ; * Registers used  :                                       *
000000010 ; * Modifications  : crctab01.asm: first program CRC generation *
000000011 ; * Description     :                                       *
000000012 ; *                                                                 *
000000013 ; * This module calculates the checksum for the CRC16 polynom. The CRC16 polynome is defined *
000000014 ; * as x16+x15+x2+x0. The calculation is done by bitwise checking. The algorithm is designed *
000000015 ; * for a two byte wide message. The algorithm can easily be modified for longer messages. The *
000000016 ; * only thing what has to be done is to check after the low byte is shifted into the high byte *
000000017 ; * that the low byte is loaded with a new data byte. The number of iteration has to be adjusted *
000000018 ; * to the number of extra bytes in the data message. The number is calculated as follows: *
000000019 ; * n=16+x*messagebits. For example if the message is 4 bytes long the number of iterations is *
000000020 ; * n=16+16bits. The extra iterations have to be done because the message is extended with 16 *
000000021 ; * zeros at the end of the message.                               *

000000001 ; **************************************************************************
000000002 ; * Title           : CRC16 calculation table driven implementation  *
000000003 ; * Author          : Thomas Schmidt                            *
000000004 ; * Date            : 22.03.1999                               *
000000005 ; * Revision        : 0.1                                     *
000000006 ; * Last Modified   : 15.04.1999                              *
000000007 ; * Core            : 12-bit, 14-bit (12-bit core tested)      *
000000008 ; * Peripherals used: none                                   *
000000009 ; * Registers used  :                                       *
000000010 ; * Modifications  : crctab01.asm: first program CRC generation *
000000011 ; * Description     :                                       *
000000012 ; *                                                                 *
000000013 ; * This module calculates the checksum for the CRC16 polynom. The CRC16 polynome is defined *
000000014 ; * as x16+x15+x2+x0. The calculation is done by bitwise checking. The algorithm is designed *
000000015 ; * for a two byte wide message. The algorithm can easily be modified for longer messages. The *
000000016 ; * only thing what has to be done is to check after the low byte is shifted into the high byte *
000000017 ; * that the low byte is loaded with a new data byte. The number of iteration has to be adjusted *
000000018 ; * to the number of extra bytes in the data message. The number is calculated as follows: *
000000019 ; * n=16+x*messagebits. For example if the message is 4 bytes long the number of iterations is *
000000020 ; * n=16+16bits. The extra iterations have to be done because the message is extended with 16 *
000000021 ; * zeros at the end of the message.                               *

000000001 ; **************************************************************************
000000002 ; * Title           : CRC16 calculation table driven implementation  *
000000003 ; * Author          : Thomas Schmidt                            *
000000004 ; * Date            : 22.03.1999                               *
000000005 ; * Revision        : 0.1                                     *
000000006 ; * Last Modified   : 15.04.1999                              *
000000007 ; * Core            : 12-bit, 14-bit (12-bit core tested)      *
000000008 ; * Peripherals used: none                                   *
000000009 ; * Registers used  :                                       *
000000010 ; * Modifications  : crctab01.asm: first program CRC generation *
000000011 ; * Description     :                                       *
000000012 ; *                                                                 *
Initialization what has to be done before CRC16 routine can be called. The FSR register contains the pointer to the first byte of data and the register DATABYTES contains the number of data bytes.

; initialization
movlw 0x10 ; set pointer to first data location
movwf FSR ; initialize FSR register
main            ; calculate CRC16 value
movf CRC_HIGH, w ; copy CRC_HIGH data into w-register
movwf INDF ; copy CRC_HIGH behind last data byte
incf FSR, f ; point to next data location
movf CRC_LOW, w ; copy CRC_LOW data into w-register
movwf INDF ; copy data into next location
movlw 0x10 ; set pointer to first data location
movwf FSR ; initialize FSR register
testPoint       ; Restore CRC
stop            ; do forever

; ******************************************************************************
; * Title:              CRC16 Table driven calculation                        *
; * Input parameter:    Pointer to first data byte (pointer in FSR register)   *
; *                     Number of data bytes stored in register DATABYTES      *
; * Output:             CRC result stored in CRC_HIGH and CRC_LOW               *
; ******************************************************************************

crc16Generate   ; initialize CRC registers
movlw 0x03 ; initialize TEMP register
movwf TEMP ; with 0x02

; check if last CRC_BUFF points to last element in table. These elements
; cannot be read from the look up table, because they are beyond the
; program memory page.
NextValueGen ; load CRC_BUFF into w-register

; cannot be read from the look up table, because they are beyond the
; program memory page.

; check if content equals to 0xff
xorlw 0xff
btfss STATUS, Z
CalculateCRC

; Most important register.
xorw CRC_BUFF
move CRC_BUFF
}
0013 0C02 00082 movlw LastTableElementHigh ; yes, get last table element for high byte
0014 01A8 00083 xorwf CRC_HIGH, f ; XOR with high byte
0015 0C02 00084 movlw LastTableElementLow ; get last table element for low byte
0016 01A7 00085 xorwf CRC_LOW, f ; XOR with low byte
0017 0A22 00086 goto DecDATABYTES ; goto end of loop
0018 0209 00088 CalculateCRC movf CRC_BUFF, w ; copy high byte of CRC into w-register
0019 04C3 00089 bcf STATUS, PA1 ; select page 1
001A 05A3 0008A bsf STATUS, PA0 ; select page 1
001B 0900 0008B call CRC16TableHigh ; get value for high byte
001C 01A8 0008C xorwf CRC_HIGH, f ; XOR table element with high byte
001D 0209 0008D movf CRC_BUFF, w ; get value for low byte
001E 05C3 0008E bcf STATUS, PA1 ; select page 2
001F 0A43 0008F bcf STATUS, PA0 ; select page 2
0020 0900 00090 call CRC16TableLow ; get value out of table
0021 01A7 00091 xorwf CRC_LOW, f ; XOR with low byte
0022 04A3 00092 DecDATABYTES bcf STATUS, PA0 ; select page 0
0023 04C3 00093 bcf STATUS, PA1 ; select page 0
0024 02EA 00094 decfsz DATABYTES, f ; decrement data bytes
0025 0A30 00095 goto ReloadGen ; reload values
0026 02EB 00096 decfsz TEMP, f ; append two bytes with zeros
0027 0A29 00097 goto AppendZeros ; append zeros to message (do twice)
0028 0800 00098 retlw 0x00 ; return to main
0029 0208 00099 AppendZeros movf CRC_HIGH, w ; copy high byte into w-register
002A 0029 0009A movf CRC_BUFF, w ; and from there to CRC_BUFF
002B 0207 0009B movf CRC_LOW, w ; Copy low byte into w-register
002C 0028 0009C movf CRC_HIGH ; and from there into CRC_HIGH
002D 0067 0009D clrf CRC_LOW ; and from there into CRC_LOW
002E 02AA 0009E incf DATABYTES, f ; increment for additional iteration
002F 0A0F 0009F goto NextValueGen ; calculate CRC for next byte
0030 0932 0009G call Reload ; reload registers
0031 0A0F 0009H goto NextValueGen ; calculate next CRC value
0032 0208 0009I ; **************************************************************************
0033 0208 0009J Reload movf CRC_HIGH, w ; copy high byte into w-register
0034 0029 0009K movf CRC_BUFF, w ; and from there to CRC_BUFF
0035 0028 0009L movf CRC_HIGH, w ; Copy low byte into w-register
0036 0200 0009M movf INDF, w ; copy next data into w-register
0037 02A4 0009N incf FSR, f ; point to next data byte
0038 0128 0009O ; **************************************************************************
003A 095E 00138 CRC16Restore call CRC16Init ; initialize CRC registers
003B 0C02 00139 movlw 0x02                    ; add two onto
003C 01EA 00140                 addwf DATABYTES, f            ; register DATABYTES
00141
00142 ; check if last CRC_BUFF points to last element in table. These elements
00143 ; cannot be read from the look up table, because they are beyond the
00144 ; program memory page.
003D 0209 00145 NextValueRes movf CRC_BUFF, w             ; load CRC_BUFF into w-register
003E 0FFF 00146 xorlw 0xff                    ; check if content equals to 0xff
00147 ; if not, calculate CRC
0040 0A46 00148 goto CalculateCRCRes ; no, calculate CRC
0041 0C02 00149 movlw LastTableElementHigh   ; yes, get last table element for high byte
0042 01A8 00150 xorwf CRC_HIGH, f             ; XOR with high byte
0043 0C02 00151 movlw LastTableElementLow    ; get last table element for low byte
0044 01A7 00152 xorwf CRC_LOW, f              ; XOR with low byte
0045 0A50 00153                 goto DecDATABYTESRes ; goto end of loop
00154
0046 0209 00155 CalculateCRCRes movf CRC_BUFF, w             ; copy high byte of CRC into w-register
0047 04C3 00156 bcf STATUS, PA1             ; select page 1
0048 05A3 00157 bsf STATUS, PA0             ; select page 1
0049 0900 00158 call CRC16TableHigh        ; get value for high byte
004A 01A8 00159 xorwf CRC_HIGH, f           ; XOR table element with high byte
004B 0209 00160 movf CRC_BUFF, w             ; get value for low byte
004C 05C3 00161 bcf STATUS, PA1             ; select page 2
004D 04A3 00162 bcf STATUS, PA0             ; select page 2
004E 0900 00163 call CRC16TableLow          ; get value out of table
004F 01A7 00164 xorwf CRC_LOW, f             ; XOR with low byte
0050 04A3 00165 DecDATABYTESRes bcf STATUS, PA0 ; select page 0
0051 04C3 00166 bcf STATUS, PA1             ; select page 0
0052 02EA 00167 decfsz DATABYTES, f         ; decrement data bytes
0053 0A5C 00168                 goto ReloadRes ; calculate next CRC16 value
00169
00170 ; check if CRC_HIGH and CRC_LOW equal to zero
0054 0228 00171 movf CRC_HIGH, f             ; copy CRC_HIGH onto itself
0055 0743 00172 btfss STATUS, Z              ; is content zero?
0056 0A5B 00173 goto CRCError                ; no, CRC error occurred
0057 0227 00174 movf CRC_LOW, f              ; copy CRC_LOW register onto itself
0058 0743 00175 btfss STATUS, Z              ; is content zero?
goto CRCError ; no, CRC error occurred
retlw 0x00 ; return to main (0= no error)
retlw 0x01 ; return to main with error code 1
goto NextValueRes ; calculate next value
movf INDF, w ; copy data into W-register
movf INDF, w ; copy data into W-register
movf INDF, w ; copy data into W-register
movf INDF, w ; copy data into W-register
movf INDF, w ; copy data into W-register
movf INDF, w ; copy data into W-register
movf INDF, w ; copy data into W-register
incf FSR, f ; set pointer to next location
incf FSR, f ; set pointer to next location
incf FSR, f ; set pointer to next location
incf FSR, f ; set pointer to next location
incf FSR, f ; set pointer to next location
movlw DataLength ; copy number of data bytes
movlw 0x03 ; decrement three from the number
movlw 0x00 ; return from subroutine
movlw 0x00 ; return to main
addwf PCL, f ; add to low byte of PC
dt 0, 0x80, 0x80, 0
dt 0x80, 0, 0, 0x80
dt 0x80, 0, 0, 0x80
dt 0, 0x80, 0x80, 0
0x80
dt 0x80, 0, 0, 0x80
0x80
dt 0x80, 0, 0, 0x80
0x80
dt 0x80, 0, 0, 0x80
0x80
02D1 0882 0802 0802 00265    dt  0x82, 0x2, 0x2, 0x82
0882
02D5 0802 0882 0882 00266    dt  0x2, 0x82, 0x82, 0x2
0802
02D9 0802 0882 0882 00267    dt  0x2, 0x82, 0x82, 0x2
0802
02DD 0882 0802 0802 00268    dt  0x82, 0x2, 0x2, 0x82
0882
02E1 0882 0802 0802 00269    dt  0x82, 0x2, 0x2, 0x82
0882
02E5 0802 0882 0882 00270    dt  0x2, 0x82, 0x82, 0x2
0802
02E9 0802 0882 0882 00271    dt  0x2, 0x82, 0x82, 0x2
0802
02ED 0802 0882 0882 00272    dt  0x2, 0x82, 0x82, 0x2
0882
02F1 0802 0802 0882 00273    dt  0x2, 0x82, 0x82, 0x2
0802
02F5 0882 0802 0802 00274    dt  0x82, 0x2, 0x2, 0x82
0882
02F9 0802 0802 0802 00275    dt  0x2, 0x82, 0x82, 0x2
0802
02FD 0802 0882 0882 00276    dt  0x2, 0x82, 0x82
00277
00278
00279    ; ********************************************************************************
00280    ; * Title: CRC16 Table for low byte                                         *
00281    ; * Input: Pointer to table element in w-register                      *
00282    ; * Output: look-up value in w-register                                *
00283
0400 00284
0400 01E2 00285 Addwf PCL, f    ; add to low byte of PC
0401 0800 0805 080F 00286    dt  0, 0x5, 0xf, 0xa
080A
0405 081B 081E 0814 00287    dt  0x1b, 0x1e, 0x14, 0x11
0811
0409 0833 0836 083C 00288    dt  0x33, 0x36, 0x3c, 0x39
0839
040D 0828 082D 0827 00289    dt  0x28, 0x2d, 0x27, 0x22
0822
0411 0863 0866 086C 00290    dt  0x63, 0x66, 0x6c, 0x69
0869
0415 0878 087D 0877 00291    dt  0x78, 0x7d, 0x77, 0x72
0872
0419 0850 0855 085F 00292    dt  0x50, 0x55, 0x5f, 0x5a
085A
041D 084B 084E 0844 00293    dt  0x4b, 0x4e, 0x44, 0x41
Program Memory Words Used: 621
Program Memory Words Free: 1427

Errors : 0
Warnings : 0 reported, 0 suppressed
Messages : 5 reported, 0 suppressed
AMERICAS
Corporate Office
Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, AZ  85224-6199
Tel:    480-786-7200  Fax: 480-786-7277
Technical Support: 480-786-7627
Web Address: http://www.microchip.com

Atlanta
Microchip Technology Inc.
500 Sugar Mill Road, Suite 200B
Atlanta, GA  30350
Tel: 770-640-0034  Fax: 770-640-0307

Boston
Microchip Technology Inc.
5 Mount Royal Avenue
Marlborough, MA  01752
Tel: 508-480-9990  Fax: 508-480-8575

Chicago
Microchip Technology Inc.
333 Pierce Road, Suite 180
Itasca, IL  60143
Tel: 630-285-0071  Fax: 630-285-0075

Dallas
Microchip Technology Inc.
4570 Westgrove Drive, Suite 160
Addison, TX  75241
Tel: 972-818-7423  Fax: 972-818-2924

Dayton
Microchip Technology Inc.
Two Prestige Place, Suite 150
Miamisburg, OH  45342
Tel: 937-291-1654  Fax: 937-291-9175

Detroit
Microchip Technology Inc.
150 Motor Parkway, Suite 202
Hauppauge, NY  11788
Tel: 631-273-5305  Fax: 631-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA  95131
Tel: 408-436-7950  Fax: 408-436-7955

AMERICAS (continued)

Toronto
Microchip Technology Inc.
5925 Airport Road, Suite 200
Mississauga, Ontario L4V 1W1, Canada
Tel: 905-405-6279  Fax: 905-405-6253

ASIA/PACIFIC
China - Beijing
Microchip Technology, Beijing
Unit 915, 6 Chaoyangmen Bei Dajie
Dong Erhuan Road, Dongcheng District
New China Hong Kong Manhattan Building
Beijing, 100027, P.R.C.
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Shanghai
Microchip Technology
Unit B701, Far East International Plaza,
No. 317, Xianxia Road
Shanghai, 200051, P.R.C.
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

Hong Kong
Microchip Asia Pacific
Unit 2101, Tower 2
Metropiazza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2-401-1200 Fax: 852-2-401-3431

India
Microchip Technology Inc.
India Liaison Office
No. 6, Legacy, Convent Road
Bangalore, 560 025, India
Tel: 91-80-229-0061 Fax: 91-80-229-0062

Japan
Microchip Technology Intl. Inc.
Benex S-1 6F
3-18-2, Shin-yokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea
Tel: 82-2-554-7200 Fax: 82-2-558-5934

ASIA/PACIFIC (continued)

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 189980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan
Microchip Technology Taiwan
10F-1C 207
Tung Hua North Road
Taipei, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE
Denmark
Microchip Technology Danmark ApS
Regus Business Centre
Lastrup høj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9985 Fax: 45 4420 9910

France
Arizona Microchip Technology SARL
Parc d’Activite du Moulin de Massy
43 Rue du Saule Trapsu
Bâtiment A - 1er Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Arizona Microchip Technology GmbH
Gustav-Heinemann-Ring 125
D-81739 München, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Arizona Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1  Fax: 39-039-6989883

United Kingdom
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnerh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5858 Fax: 44-118 921-5835

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999. The Company’s quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, Keil® code hopping devices, Serial EEPROMs and microperipheral products. In addition, Microchip’s quality system for the design and manufacture of development systems is ISO 9001 certified.

All rights reserved. © 2000 Microchip Technology Incorporated. Printed in the USA. 5/00 Printed on recycled paper.