Section 60. 32-Bit Programmable Cyclic Redundancy Check (CRC)

HIGHLIGHTS

This section of the manual contains the following major topics:

Note:	This family reference manual section is meant to serve as a complement to device data sheets. Depending on the device variant, this manual section may not apply to all PIC32 devices. Please consult the note at the beginning of the **"32-Bit Programmable Cyclic Redundancy Check (CRC) Generator"** chapter in the current device data sheet to check whether this document supports the device you are using. Device data sheets and family reference manual sections are available for download from the Microchip Worldwide Web site at: http://www.microchip.com

60.1 INTRODUCTION

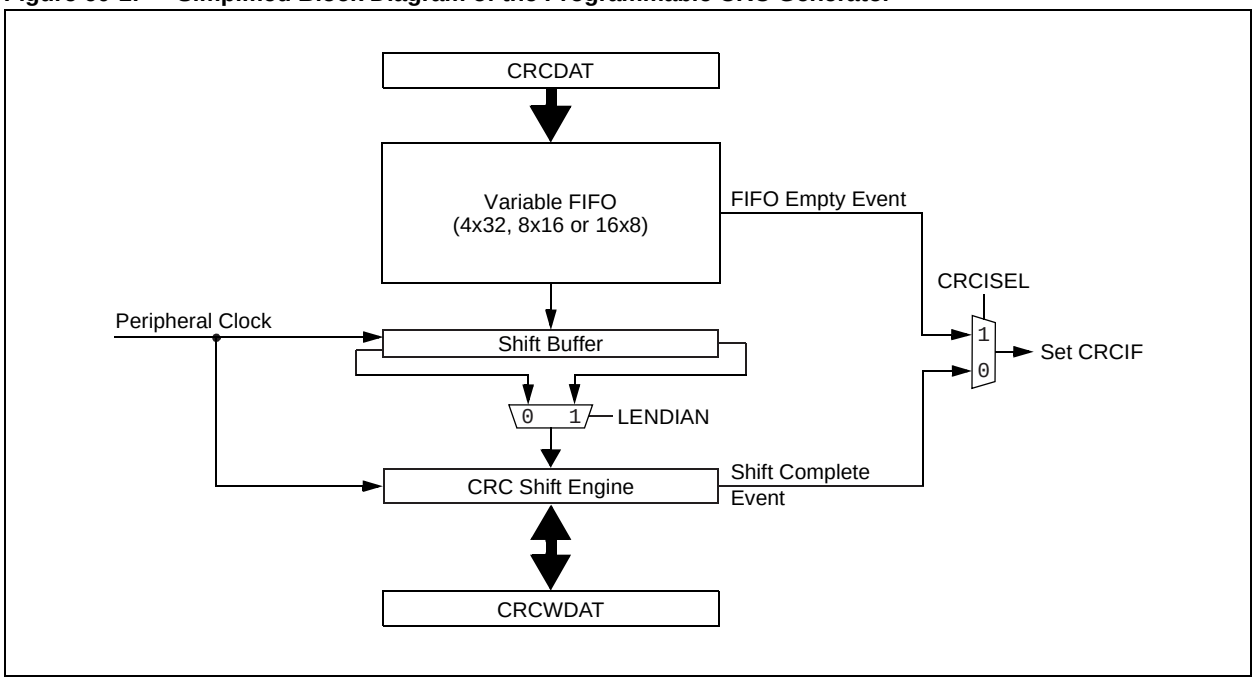
The 32-Bit Programmable Cyclic Redundancy Check (CRC) module is a software-configurable CRC generator. The module provides a hardware implemented method of quickly generating checksums for various communication and security applications. The CRC engine calculates the CRC checksum without CPU intervention; moreover, it is much faster than the software implementation.

The programmable CRC generator provides the following features:

- User-programmable CRC polynomial equation, up to 32 bits
- Programmable shift direction (little or big-endian)
- Independent data and polynomial lengths
- Configurable interrupt output
- Data FIFO

The programmable CRC generator module can be divided into two parts: the control logic and the CRC engine. The control logic incorporates a register interface, data FIFO, an interrupt generator and a CRC engine interface. The CRC engine incorporates a CRC calculator, which is implemented using a serial shifter with XOR function. A simplified block diagram is shown in Figure 60-1.

Figure 60-1: Simplified Block Diagram of the Programmable CRC Generator



60.2 CRC OVERVIEW

The checksum is a unique number associated with a message, or a particular block of data, containing several bytes. Whether it is a data packet for communication, or a block of data stored in memory, a piece of information, such as a checksum, helps to validate it before processing. The simplest way to calculate a checksum is to add together all the data bytes present in the message. However, this method of checksum calculation fails badly when the message is modified by inverting or swapping groups of bytes. Also, it fails when null bytes are added anywhere in the message.

The Cyclic Redundancy Checksum (CRC) is a more complicated, but robust, error checking algorithm. The main idea behind the CRC algorithm is to treat a message as a binary bit stream and divide it by a fixed binary number. The remainder from this division is considered to be the checksum. Similar to division, the CRC calculation is also an iterative process. The only difference is that these operations are done on modulo arithmetic, based on mod 2. For example, division is replaced with the XOR operation (i.e., subtraction without carry). The CRC algorithm uses the term, polynomial, to perform all of its calculations. The divisor, dividend and remainder that are represented by numbers are termed as: polynomials with binary coefficients. For example, the number, 25h (11001), is represented as:

Equation 60-1:

$$(1 * x^4) + (1 * x^3) + (0 * x^2) + (0 * x^1) + (1 * x^0) \text{ or } x^4 + x^3 + x^0$$

In order to perform the CRC calculation, a suitable divisor is first selected. This divisor is called the generator polynomial. Since CRC is used to detect errors, a generator polynomial of a suitable length needs to be chosen for a given application, as each polynomial has different error detection capabilities. Some polynomials are widely used for many applications, but the error detecting capabilities of any particular polynomial are beyond the scope of this reference section.

The CRC algorithm is straightforward to implement in software. However, it requires considerable CPU bandwidth to implement the basic requirements, such as shift, bit test and XOR. Moreover, CRC calculation is an iterative process and additional software overhead for data transfer instructions puts an enormous burden on the MIPS requirement of a microcontroller. In contrast, the software-configurable CRC hardware module facilitates a fast CRC checksum calculation with minimal software overhead.

60.3 REGISTERS

The CRC module uses the following Special Function Registers (SFRs):

- **CRCCON: CRC Control Register**
  This register controls all of the module's functions, including data and polynomial size, Accumulator and Endian modes, and interrupt generation.
- **CRCXOR: CRC XOR Register**
  This register configures the CRC polynomial.
- **CRCDAT: CRC FIFO Data Register (write-only)**
  The register loads initial data for the calculation.
- **CRCWDAT: CRC Shift Data Register**
  This register loads the initial CRC value and holds the final result when the calculation is done.

Table 60-1 and Register 60-1 through Register 60-4 contain the description of the CRC module registers.

Table 60-1: CRC SFR Summary(1)

Name	Bit Range	Bit 31/15	Bit 30/14	Bit 29/13	Bit 28/12	Bit 27/11	Bit 26/10	Bit 25/9	Bit 24/8	Bit 23/7	Bit 22/6	Bit 21/5	Bit 20/4	Bit 19/3	Bit 118/2	Bit 17/1	Bit 16/0
CRCCON	31:16	—	—	—	DWIDTH<4:0>					—	—	—	PLEN<4:0>				
	15:0	ON	—	SIDL	VWORD<4:0>					CRCFUL	CRCMPT	CRCISEL	CRCGO	LENDIAN	MOD	—	—
CRCXOR	31:16	X<31:16>															
	15:0	X<15:0>															
CRCDAT	31:16	DATA<31:16>															
	15:0	DATA<15:0>															
CRCWDAT	31:16	SDATA<31:16>															
	15:0	SDATA<15:0>															

Note 1: All registers have associated Clear, Set and Invert registers, suffixed as -CLR, -SET and -INV, at address offsets of 0x4, 0x8 and 0xC bytes, respectively. These associated registers are used to modify their main registers. Writing a '1' to a bit position in an associated register will clear, set or invert the valid corresponding bit in the main register. Reads from these registers should be ignored.

Register 60-1: CRCCON: CRC Control Register

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	DWIDTH<4:0>				
23:16	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	—	—	—	PLEN<4:0>				
15:8	R/W-0	U-0	R/W-0	R-0	R-0	R-0	R-0	R-0
	ON	—	SIDL	VWORD<4:0>				
7:0	R-0	R-1	R/W-0	R/W-0	R/W-0	R/W-0	U-0	U-0
	CRCFUL	CRCMPT	CRCISEL	CRCGO	LENDIAN	MOD	—	—

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 31-29 **Unimplemented:** Read as '0'

bit 28-24 **DWIDTH<4:0>:** Data Word Width Configuration bits
Configures the width of the data word (Data Word Width – 1).

bit 23-21 **Unimplemented:** Read as '0'

bit 20-16 **PLEN<4:0>:** Polynomial Length Configuration bits
Configures the length of the CRC polynomial (Polynomial Length – 1).

bit 15 **ON:** CRC Enable bit
1 = Module is enabled
0 = Module is disabled

bit 14 **Unimplemented:** Read as '0'

bit 13 **SIDL:** CRC Stop in Idle Mode bit
1 = Module operation discontinues when device enters Idle mode
0 = Module operation continues in Idle mode

bit 12-8 **VWORD<4:0>:** Counter Value bits
Indicates the number of unprocessed words in the FIFO buffer; buffer depth is determined by the data word size.

bit 7 **CRCFUL:** CRC FIFO Full bit
1 = FIFO is full
0 = FIFO is not full

bit 6 **CRCMPT:** CRC FIFO Empty bit
1 = FIFO is empty
0 = FIFO is not empty

bit 5 **CRCISEL:** CRC Interrupt Selection bit
1 = Generates an interrupt when the FIFO is empty (final word of data is still shifted through CRC)
0 = Generates an interrupt on a shift complete (FIFO is empty and no data is shifted from the shift buffer)

bit 4 **CRCGO:** Start CRC Shift bit
1 = Starts CRC serial shifter; clearing the bit aborts shifting
0 = CRC serial shifter is turned off

bit 3 **LENDIAN:** Data Word Little-Endian Configuration bit
1 = Data word is shifted into the CRC, starting with the LSb (little-endian); reflected input data
0 = Data word is shifted into the CRC, starting with the MSb (big-endian); non-reflected input data

bit 2 **MOD:** CRC Operating Mode Select bit
1 = Alternate mode: Shift buffer data is XORed with CRC shift engine after bit n
0 = Legacy mode: Shift buffer data is XORed with CRC shift engine before bit 0

bit 1-0 **Unimplemented:** Read as '0'

Register 60-2: CRCXOR: CRC XOR Register

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	X<31:24>							
23:16	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	X<23:16>							
15:8	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	X<15:8>							
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	X<7:0>							

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 31-0 **X<31:0>:** XOR of Polynomial Term x^n Enable bits
1 = Polynomial term x^n is used in CRC polynomial
0 = Polynomial term is not used

Register 60-3: CRCDAT: CRC FIFO Data Register (write-only)

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0
	DATA<31:24>							
23:16	W-0	W-0	W-0	W-0	W-0	W-0	W-0	W-0
	DATA<23:16>							
15:8	W-0	R/W-0	W-0	W-0	W-0	W-0	W-0	W-0
	DATA<15:8>							
7:0	W-0	R/W-0	W-0	W-0	W-0	W-0	W-0	W-0
	DATA<7:0>							

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 31-0 **DATA<31:0>:** FIFO Data bits
Data written to this register is written to the next open location in the CRC FIFO buffer.
Reading this register returns a value of '0'.

Register 60-4: CRCWDAT: CRC Shift Data Register

Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
31:24	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	SDATA<31:24>							
23:16	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	SDATA<23:16>							
15:8	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	SDATA<15:8>							
7:0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	SDATA<7:0>							

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 31-0 **SDATA<31:0>:** Shift Register Data bits

Writing to this register writes directly to the CRC Engine Shift register, overwriting any current value.
Reading this register returns the current value of the CRC Engine Shift register.

60.4 CRC ENGINE

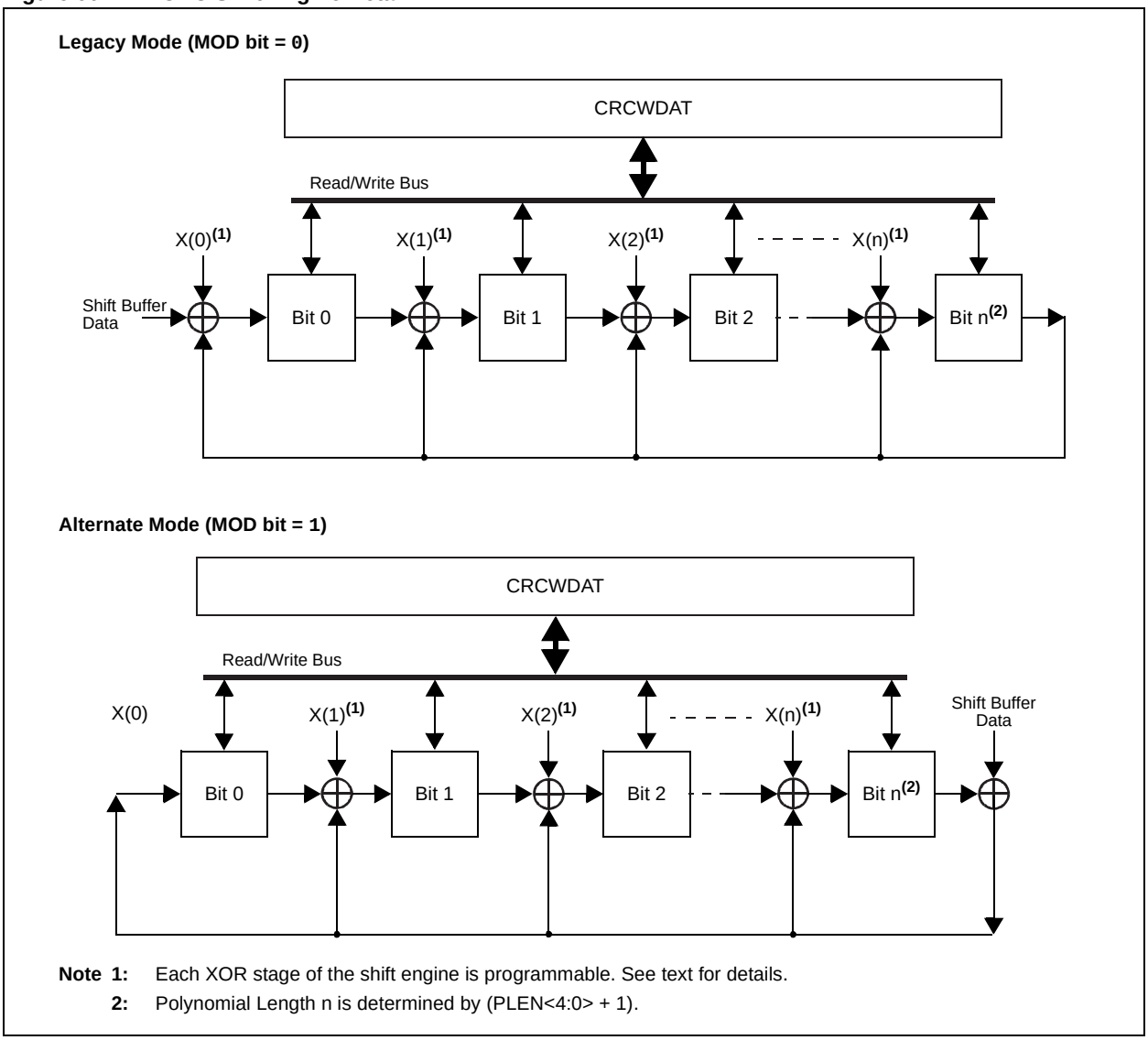
60.4.1 Generic CRC Engine

The CRC engine is a serial shifting CRC calculator, configurable though multiplexer settings. The engine can also be configured as to where shift buffer data is introduced using the MOD bit (CRCCON<2>). A simplified diagram of the CRC shift engine is shown in Figure 60-2.

The CRC algorithm uses a simplified form of arithmetic process, using the XOR operation instead of binary division. The coefficients of the generator polynomial are programmed with the CRCXOR<31:1> bits. Writing a '1' into a location enables XORing of that element in the polynomial. The length of the polynomial is programmed using the PLEN<4:0> bits in the CRCCON register (CRCCON<20:16>). The value of PLEN<4:0> signals the length of the polynomial and switches a multiplexer to indicate the tap from which the feedback originated.

The result of the CRC calculation is obtained by reading the CRCWDAT register.

Figure 60-2: CRC Shift Engine Detail



Note 1: Each XOR stage of the shift engine is programmable. See text for details.

2: Polynomial Length n is determined by (PLEN<4:0> + 1).

60.5 CONTROL LOGIC

60.5.1 Polynomial Interface

The CRC module can be programmed for CRC polynomials of up to the 32nd order, using up to 32 bits. Polynomial length, which reflects the highest exponent in the equation, is selected by the PLEN<4:0> bits (CRCCON<20:16>). The CRCXOR register controls which exponent terms are included in the equation. Setting a particular bit includes that exponent term in the equation functionally; this includes an XOR operation on the corresponding bit in the CRC engine. Clearing the bit disables the XOR.

For example, consider two CRC polynomials: one a 16-bit equation and the other a 32-bit equation (Equation 60-2).To program these polynomials into the CRC generator, set the register bits as shown in Table 60-2.

Equation 60-2:

$$x^{16} + x^{12} + x^5 + 1$$

and

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Table 60-2: CRC Setup Examples for 16 and 32-Bit Polynomials

CRC Control Bits	Bit Values	
	16-Bit Polynomial	32-Bit Polynomial
PLEN<4:0>	01111	11111
X<31:16>	0000 0000 0000 0000	0000 0100 1100 0001
X<15:1>	0001 0000 0010 0001	0001 1101 1011 0111

Note that the appropriate positions are set to ‘1’ to indicate that they are used in the equation (e.g., X26 and X23). The Most Significant bit (MSb) of the polynomial does not affect the calculation and can be set to any value.

60.5.2 Data Shift Direction

The LENDIAN bit (CRCCON<3>) is used to control the shift direction. By default, the CRC module will shift data through the engine, MSb first (LENDIAN = 0). Setting LENDIAN to ‘1’ causes the CRC module to shift data, LSb first. This setting allows better integration with various communication schemes and removes the overhead of reversing the bit order in software. Note that this only changes the direction that the data is shifted into the engine. The result of the CRC calculation will still be a normal CRC result, not a reverse CRC result.

PIC32 devices are little-endian. When the CRC module is configured for the big-endian (LENDIAN = 0), the input data bytes and words must be swapped in the application code before loading them into the data FIFO (CRCDAT register).

60.5.3 Data FIFO

The module incorporates a FIFO that works with a variable data width. The data width is defined by the DWIDTH<4:0> bits (CRCCON<28:24>). It can be configured to any value, between 1 and 32 bits. The logic associated with the FIFO contains a 5-bit counter, the VWORD<4:0> bits (CRCCON<12:8>).

The value in the VWORD<4:0> bits indicates the number of unprocessed data elements in the FIFO. The FIFO is:

- 16-word deep when DWIDTH<4:0> ≤ 7 (data words, 8-bit wide or less)
- 8-word deep when DWIDTH<4:0> ≤ 15 (data words from 9 to 16-bit wide)
- 4-word deep when DWIDTH<4:0> ≤ 31 (data words from 17 to 32-bit wide)

The data for the CRC calculation must be written into the FIFO using the CRCDAT register. Reading CRCDAT always returns zero. To accommodate the MSb first shift method (LENDIAN = 0), byte and word swapping must be done in software when filling the FIFO.

Note:	Ensure that the new data is not written into the CRCDAT register when the CRCFUL bit is set; if the new data is written, it will be ignored.

When all shifts are done (i.e., the FIFO is empty and the CRC shift engine is Idle), it is possible to change the FIFO width (DWIDTH<4:0> bits) without any information loss or CRC result damage.

With a data width of 8 bits or less, the FIFO increments on a write to the lower byte of the CRCDAT register (a byte access to the CRCDAT register must be used). The smallest data element that can be written into the FIFO is 1 byte.

For example, if DWIDTH<4:0> is 5, then the size of the data is DWIDTH<4:0> + 1 or 6. The data is written as a whole byte; the two unused upper bits are ignored. Once the data byte is written into the CRCDAT register, the value of the VWORD<4:0> bits (CRCCON<12:8>) increments by one.

With data widths more than 8 bits, and less than or equal to 16 bits, the FIFO increments on a write to the lower 16-bit word of the CRCDAT register (16-bit word access to the CRCDAT register must be used). Unused upper data bits are ignored. The value of the VWORD<4:0> bits is incremented for every write to the CRCDAT register.

When the data width is greater than 16 bits, any write to the CRCDAT register increments the VWORD<4:0> bits by one. Unused upper data bits are ignored.

60.5.4 CRC Engine Interface

60.5.4.1 FIFO TO CRC SHIFT ENGINE

To start moving the data from the FIFO to the CRC shift buffer, the CRCGO bit (CRCCON<4>) must be set. The serial shifter starts shifting data from the shift buffer to the CRC shift engine, starting from the MSb first for LENDIAN = 0, and LSb first for LENDIAN = 1, when CRCGO = 1 and the value of VWORD<4:0> is greater than zero. If the CRCFUL bit was set earlier, then it is cleared when the VWORDx bits decrement by one. The VWORD<4:0> bits decrement by one when a FIFO location is moved to the shift buffer. The serial shifter continues shifting until the VWORD<4:0> bits reach zero; at this point, the CRCMPT bit becomes set to indicate that the FIFO is empty. If the CRCGO bit is cleared during a CRC calculation, then the CRC shift engine will stop calculating until the CRCGO bit is set.

The application can write into the FIFO while the shift operation is in progress. The CRCFUL bit should be monitored. If the CRCFUL bit is not set, another word can be written into the FIFO. At least one instruction cycle must pass after a write to the CRCDAT register, before a read of the valid value of the VWORD<4:0> bits.

When the VWORD<4:0> bits reach the maximum value for the configured value of the DWIDTH<4:0> bits, the CRCFUL bit becomes set. When the VWORD<4:0> bits reach zero, the CRCMPT bit becomes set. The FIFO is emptied and the VWORD<4:0> bits are set to '00000' whenever ON is '0'.

60.5.4.2 NUMBER OF CLOCK CYCLES TO SHIFT DATA

The data from the FIFO goes to the shift buffer. It takes 2 peripheral clock cycles to start moving the data words from the FIFO to the shift buffer. The data from the shift buffer is then shifted to the CRC shift engine. It takes (DWIDTH<4:0> + 1) clock cycles to completely move the data from the shift buffer to the CRC shift engine. For example, if DWIDTH<4:0> = 5, then the data length is 6 bits (DWIDTH<4:0> + 1) and 6 cycles are required to shift the data. In this case, only 6 bits of a byte are shifted out. The two MSbs of each byte are 'don't care' bits. Similarly, for a 12-bit polynomial selection, the Most Significant 4 bits of each word are ignored.

60.5.4.3 CRC INITIAL VALUE

The access to the CRC shift engine is provided through the CRCWDAT register. This register can be loaded with a desired CRC initial value prior to the start of the calculations. The form of this initial value depends on the operating mode selected by the MOD bit (CRCCON<2>).

In Alternate mode (MOD = 1), the CRC initial value must be in direct form.

In Legacy mode (MOD = 0), the CRC initial value must be in non-direct form. The non-direct initial value is a value for which the CRC calculation gives the desired direct CRC initial value. For example, if the application uses the CRC-32 polynomial, 0x04C11DB7, and must start the calculations from the CRC direct initial value, 0xFFFFFFFF, then the non-direct value, 0x46AF6449, must be loaded in the CRCWDAT register (the CRC of this non-direct value, 0x46AF6449, is 0xFFFFFFFF). When the non-direct initial value is written into the shift engine using the CRCWDAT register, it will be converted by the CRC module to the direct initial value after (PLEN<4:0> + 1) peripheral clock cycles.

Note:	The write to the CRCWDAT register clears/resets the shift buffer.

Usually the CRC calculation starts from the same initial value every time. In this case, the non-direct initial value can be found just once and then can be defined as a constant in the application code.

Note:	The CRC non-direct initial value of zero is zero.

Example 60-1 shows a possible software routine to get the non-direct initial value from the direct initial value.

Example 60-1: Software Routine to Calculate the Non-Direct Initial Value

```
unsigned   long   CalculateNonDirectSeed(
unsigned   long   seed,                     // direct CRC initial value
unsigned   long   polynomial,               // polynomial
unsigned   char   polynomialOrder)          // polynomial order
{
unsigned   char   lsb;
unsigned   char   i;
unsigned   long   msbmask;

    msbmask = ((unsigned long)1)<<(polynomialOrder-1);
    for (i=0; i<polynomialOrder; i++) {
        lsb = seed & 1;
        if (lsb) seed ^= polynomial;
        seed >>= 1;
        if (lsb) seed |= msbmask;
    }
return     seed;                            // return the non-direct CRC initial value
}
```

The CRC module can be used to get the non-direct initial value. To do this:

1. Enable the CRC module (ON = 1) and shifts (CRCGO = 1).
2. Shift the polynomial value right by one.
3. Reverse the bit order of the shifted polynomial value.
4. Write this result in the CRCXOR register.
5. Set the data width and polynomial length (DWIDTH<4:0> and PLEN<4:0> bits) to the polynomial order (length).
6. Reverse the bit order of the desired direct initial value.
7. Write the reversed initial value in the CRCWDAT register.
8. Write a dummy data to the CRCDAT register and wait 2 peripheral clock cycles to move the data from the FIFO to the shift buffer, and (PLEN<4:0> + 1) peripheral clock cycles to shift out the result.

   Alternatively, clear the CRC Interrupt Selection bit (CRCISEL = 0) to get the interrupt when shifts from the shift buffer are done, clear the CRC interrupt flag, write a dummy data in the CRCDAT register and wait for the CRC interrupt flag to set.
9. Read the value from the CRCWDAT register.
10. Reverse the bit order of the read result; it will give the final non-direct initial value.

Example 60-2 shows one way to implement this procedure.

To continue calculations of the full data message, in the applications where the intermediate CRC sums must be read in the middle of the calculations, the non-direct value must be calculated and set to the CRCWDAT register again. In this case, the CRC direct initial value will be an intermediate CRC result read.

Example 60-2: Calculating the Non-Direct Initial Value (MOD Bit = 0)

```
unsigned    long    CalculateNonDirectSeed(unsigned long seed, // direct CRC initial value
                    unsigned long polynomial,                  // polynomial
unsigned    char    polynomialOrder)                           // polynomial order (valid values
                                                               // are 8, 16, 32 bits)
{
    CRCCON = 0;
    CRCCONbits.ON = 1;                                         // enable CRC
    CRCCONbits.CRCISEL = 0;                                    // interrupt when all shifts are done
    CRCCONbits.DWIDTH = polynomialOrder-1;                     // data width
    CRCCONbits.PLEN = polynomialOrder-1;                       // polynomial length
    CRCCONbits.CRCGO = 1;                                      // start CRC calculation

    polynomial >>= 1;                                          // shift the polynomial right
    polynomial = ReverseBitOrder(polynomial, polynomialOrder); // reverse bits order of the
                                                               // polynomial
    CRCXOR = polynomial;                                       // set the reversed polynomial
    seed = ReverseBitOrder(seed, polynomialOrder);             // reverse bits order of seed value
    CRCWDAT = seed;                                            // set seed value
    IFS0CLR = _IFS0_CRCIF_MASK;                                // clear interrupt flag
    switch(polynomialOrder)                                    // load dummy data to shift out
                                                               // seed result
    {
        case 8:
             *((unsigned char*)&CRCDAT) = 0;                   // load byte
             while(!IFS0bits.CRCIF);                           // wait until shifts are done
             seed = CRCWDAT&0x0FFFF;                           // read reversed seed
        case 16:
             *((unsigned short*)&CRCDAT) = 0;                  // load short
             while(!IFS0bits.CRCIF);                           // wait until shifts are done
             seed = CRCWDAT&0x00FFFF;                          // read reversed seed
             break;
        case 32:
             // load long
             CRCDAT = 0;
             while(!IFS0bits.CRCIF);                           // wait for shifts are done
             seed = CRCWDAT;                                   // read reversed seed
             break;
        default:
             ;
    }
    seed = ReverseBitOrder(seed, polynomialOrder);             // reverse the bit order to get
                                                               // the non-direct seed
    return seed;                                               // return non-direct CRC initial value
}
```

Example 60-2: Routine to Calculate the Non-Direct Initial Value (MOD Bit = 0) (Continued)

```
 // WHERE THE FUNCTION TO REVERSE THE BIT ORDER CAN BE

unsigned long  ReverseBitOrder(unsigned long data,       // input data
unsigned char numberOfBits)                              // width of the input data,
                                                         // valid values are 8,16,32 bits
{
unsigned long  maskin  = 0;
unsigned long  maskout = 0;
unsigned long  result = 0;
unsigned char  i;

    switch(numberOfBits)
    {
        case 8:
           maskin = 0x80;
           maskout = 0x01;
           break;
        case 16:
           maskin = 0x8000;
           maskout = 0x0001;
           break;
        case 32:
           maskin = 0x80000000;
           maskout = 0x00000001;
           break;
        default:
           ;
    }
    for(i=0; i<numberOfBits; i++)
    {
        if(data&maskin){
           result |= maskout;
        }
           maskin  >>= 1;
           maskout <<= 1;
    }
    return result;
}
```

60.5.4.4 CRC RESULT

Reading the result of a CRC calculation depends on the selected operating mode.

In Alternate mode (MOD = 1), the result is available in the CRCWDAT register when all the data in the CRC FIFO buffer has been processed. Submitting dummy data to generate extra cycles is not required.

In Legacy mode (MOD = 0), the CRC module requires (PLEN<4:0> + 1) extra peripheral clock cycles to finish the calculations. To generate these additional cycles, the dummy data, with the width equal to the polynomial order (length), must be loaded into the CRCDAT register. After the shifts are finished, the final CRC result can be read from the CRCWDAT register (the CRCWDAT register provides the direct access to the CRC Shift register).

When the CRC result is achieved, the CRC non-direct initial value should be written again into the CRCWDAT register to clear/reset the shift buffer from the previously loaded dummy data to start a new calculation.

60.5.5 Interrupt Operation

The module generates an interrupt that is configurable by the user for either of the two conditions. If CRCISEL is '1', an interrupt is generated when the VWORD<4:0> bits make a transition from a value of '1' to '0'. If CRCISEL is '0', an interrupt will be generated when the FIFO is empty and shifts from the shift buffer are finished. Some CRC modules can use the persistent interrupt (or level interrupt). For these modules, it is not possible to clear the interrupt flag if the data FIFO is empty. The interrupt flag should be cleared after some data is written into the FIFO.

For more details on interrupts and interrupt priority settings, refer to **Section 8. "Interrupts"** (DS60001108) in the *"PIC32 Family Reference Manual"*.

60.6 APPLICATION OF CRC MODULE

The CRC is a robust error checking algorithm in digital communication for messages containing several bytes or words. After calculation, the checksum is appended to the message and transmitted to the receiving station. The receiver calculates the checksum with the received message to verify the data integrity.

60.6.1 Variations

The 32-bit programmable CRC module can be programmed to shift out either the MSb or LSb first. MSb first is a popular implementation as employed in XMODEM protocol. In one of the variations (CCITT protocol) for CRC calculation, the LSb is shifted out first. Discussions on all the variations are beyond the scope of this document, but several variations of CRC can be implemented using this module.

The choice of the polynomial length, and the polynomial itself, are application-dependent. Polynomial lengths of 5, 7, 8, 10, 12, 16 and 32 are normally used in various standard implementations. The following sections explain the recommended step-by-step procedure for CRC calculation. Users can decide whether zeros, or any other values, need to be appended to the message stream. Depending on the application, the user may decide whether any value needs to be appended at all.

60.6.2 Typical Operation

To use the module for a typical CRC calculation:

1. Set the ON bit to enable the module.
2. Configure the module for the desired operation:
   a) Program the desired polynomial using the CRCXOR register and the PLEN<4:0> bits.
   b) Configure the data width and shift direction using the DWIDTH<4:0> and LENDIAN bits.
3. Set the CRCGO bit to start the calculations.
4. Set the desired CRC initial value in the CRCWDAT register as described in **Section 60.5.4.3 “CRC Initial Value”**.
5. Load all data into the FIFO by writing to the CRCDAT register as space becomes available (the CRCFUL bit must be zero before the next data loading).
6. Wait until the data FIFO is empty (CRCMPT bit is set).
7. Read the CRC result as described in **Section 60.5.4.4 “CRC Result”**.

60.6.3 Application Examples for CRC Modules with Edge Interrupt

Example 60-3 through Example 60-12 show typical code for different combinations of polynomial length, data width, shift direction and CRC Engine modes for modules with edge interrupt.

Example 60-3: CRC-SMBus (8-Bit Polynomial with 32-Bit Data, Big-Endian, MOD Bit = 1)

```
// This macro is used to swap bytes for big endian
#define   Swap(x) __extension__({ \
unsigned  long __x = (x), __v; \
__asm__   ("wsbh %0,%1;\n\t" \
"rotr %0,16" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})

// ASCII bytes "12345678"
volatile  unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
volatile  unsigned char crcResultCRCSMBUS = 0;
int       main (void)
{
unsigned  long*  pointer;
unsigned  short length;
unsigned  long  data;

// standard CRC-SMBUS

#define   CRCSMBUS_POLYNOMIAL ((unsigned long)0x00000007)
#define   CRCSMBUS_SEED_VALUE ((unsigned long)0x00000000)   // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                     // alternate mode
    CRCCONbits.ON = 1;                                      // enable CRC
    CRCCONbits.LENDIAN = 0;                                 // big endian
    CRCCONbits.CRCISEL = 0;                                 // interrupt when all shifts are done
    CRCCONbits.DWIDTH = 32-1;                               // 32-bit data width
    CRCCONbits.PLEN = 8-1;                                  // 8-bit polynomial order
    CRCXOR = CRCSMBUS_POLYNOMIAL;                           // set polynomial
    CRCWDAT = CRCSMBUS_SEED_VALUE;                          // set initial value
    CRCCONbits.CRCGO = 1;                                   // start CRC calculation

    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                           // wait if FIFO is full
        data = *pointer++;                                  // load from little endian
        data = Swap(data);                                  // swap bytes for big endian
        length--;
        if(length == 0)
        {
          break;
        }
        CRCDAT = data;                                      // 32-bit word access to FIFO
    }
    CRCCONbits.CRCGO = 0;                                   // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                             // clear the interrupt flag
    CRCDAT = data;                                          // write last data into FIFO
    CRCCONbits.CRCGO = 1;                                   // resume CRC calculation

    while(!IFS0bits.CRCIF);                                 // wait until shifts are done
    crcResultCRCSMBUS = (unsigned char)CRCWDAT&0x00ff;      // get CRC result (must be 0xC7)

    while(1);
    return 1;
}
```

Example 60-4: CRC-SMBus (8-Bit Polynomial with 32-Bit Data, Little-Endian, MOD Bit = 0)

```
// This macro is used to swap bytes for big endian
#define   Swap(x) __extension__({ \
unsigned  long __x = (x), __v; \
__asm__   ("wsbh %0,%1;\n\t" \
"rotr %0,16" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})

// ASCII bytes "12345678"
volatile  unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
volatile  unsigned char crcResultCRCSMBUS = 0;
int       main (void)
{
unsigned  long* pointer;
unsigned  short length;
unsigned  long  data;

// standard CRC-SMBUS

#define   CRCSMBUS_POLYNOMIAL ((unsigned long)0x00000007)
#define   CRCSMBUS_SEED_VALUE ((unsigned long)0x00000000) // non-direct initial value of zero

    CRCCON = 0;
    CRCCONbits.MOD = 0;                                   // legacy mode
    CRCCONbits.ON = 1;                                    // enable CRC
    CRCCONbits.LENDIAN = 0;                               // big endian
    CRCCONbits.CRCISEL = 0;                               // interrupt when all shifts are done
    CRCCONbits.DWIDTH = 32-1;                             // 32-bit data width
    CRCCONbits.PLEN = 8-1;                                // 8-bit polynomial order
    CRCXOR = CRCSMBUS_POLYNOMIAL;                         // set polynomial
    CRCWDAT = CRCSMBUS_SEED_VALUE;                        // set initial value
    CRCCONbits.CRCGO = 1;                                 // start CRC calculation
    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                         // wait if FIFO is full
        data = *pointer++;                                // load from little endian
        data = Swap(data);                                // swap bytes for big endian
        length--;
        if(length == 0)
        {
          break;
        }
        CRCDAT = data;                                    // 32-bit word access to FIFO
    }

    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear the interrupt flag
    CRCDAT = data;                                        // write last data into FIFO
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    CRCCONbits.DWIDTH = 8-1;                              // switch data width to polynomial length 8-bit
    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear the interrupt flag
    *((unsigned char*)&CRCDAT) = 0;                       // 8-bit dummy data to shift out the CRC result
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    crcResultCRCSMBUS = (unsigned char)CRCWDAT&0x00ff;    // get CRC result (must be 0xC7)

    while(1);
    return 1;
}
```

Example 60-5: CRC-16 (16-Bit Data with 32-Bit Polynomial, Little-Endian, MOD Bit = 1)

```
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRC16 = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-16

#define   CRC16_POLYNOMIAL ((unsigned long)0x00008005)
#define   CRC16_SEED_VALUE ((unsigned long)0x00000000)    // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                   // alternate mode
    CRCCONbits.ON = 1;                                    // enable CRC
    CRCCONbits.CRCISEL = 0;                               // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                               // little endian
    CRCCONbits.DWIDTH = 16-1;                             // 16-bit data width
    CRCCONbits.PLEN = 16-1;                               // 16-bit polynomial order
    CRCXOR = CRC16_POLYNOMIAL;                            // set polynomial
    CRCWDAT = CRC16_SEED_VALUE;                           // set initial value
    CRCCONbits.CRCGO = 1;                                 // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                         // wait if FIFO is full
        data = *pointer++;                                // load data
        length--;
        if(length == 0)
        {
          break;
        }
        *((unsigned short*)&CRCDAT) = data;               // 16-bit word access to FIFO
    }

    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear the interrupt flag
    *((unsigned short*)&CRCDAT) = data;                   // write last data into FIFO
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    crcResultCRC16 = (unsigned short)CRCWDAT;             // get CRC result (must be 0xE716)

    while(1);
    return 1;
}
```


Example 60-6: CRC-16 (16-Bit Data, 16-Bit Polynomial, Little-Endian, MOD Bit = 0)

```
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRC16 = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-16

#define  CRC16_POLYNOMIAL ((unsigned long)0x00008005)
#define  CRC16_SEED_VALUE ((unsigned long)0x00000000)    // non-direct initial value of zero

    CRCCON = 0;
    CRCCONbits.MOD = 0;                                  // legacy mode
    CRCCONbits.ON = 1;                                   // enable CRC
    CRCCONbits.CRCISEL = 0;                              // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                              // little endian
    CRCCONbits.DWIDTH = 16-1;                            // 16-bit data width
    CRCCONbits.PLEN = 16-1;                              // 16-bit polynomial order
    CRCXOR = CRC16_POLYNOMIAL;                           // set polynomial
    CRCWDAT = CRC16_SEED_VALUE;                          // set initial value
    CRCCONbits.CRCGO = 1;                                // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                        // wait if FIFO is full
        data = *pointer++;                               // load data
        *((unsigned short*)&CRCDAT) = data;              // 16-bit word access to FIFO
    }

    while(CRCCONbits.CRCFUL);                            // wait if FIFO is full
    CRCCONbits.CRCGO = 0;                                // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                          // clear the interrupt flag
    *((unsigned short*)&CRCDAT) = 0;                     // 16-bit dummy data to shift out CRC result
    CRCCONbits.CRCGO = 1;                                // resume CRC calculation
    while(!IFS0bits.CRCIF);                              // wait until shifts are done
    crcResultCRC16 = (unsigned short)CRCWDAT;            // get CRC result (must be 0xE716)

    while(1);
    return 1;
}
```

Example 60-7: CRC-CCITT (16-Bit Polynomial with 16-Bit Data, Big-Endian, MOD Bit = 1)

```
// This macro is used to swap bytes for big endian
#define  Swap(x) __extension__({ \
unsigned long __x = (x), __v; \
__asm__  ("wsbh %0,%1;\n\t" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRCCCITT = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-CCITT

#define  CRCCCITT_POLYNOMIAL ((unsigned long)0x00001021)
#define  CRCCCITT_SEED_VALUE ((unsigned long)0x0000FFFF)     // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                     // alternate mode
    CRCCONbits.ON = 1;                                      // enable CRC
    CRCCONbits.CRCISEL = 0;                                 // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 0;                                 // big endian
    CRCCONbits.DWIDTH = 16-1;                               // 16-bit data width
    CRCCONbits.PLEN = 16-1;                                 // 16-bit polynomial order
    CRCXOR = CRCCCITT_POLYNOMIAL;                           // set polynomial
    CRCWDAT = CRCCCITT_SEED_VALUE;                          // set initial value
    CRCCONbits.CRCGO = 1;                                   // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                           // wait if FIFO is full
        data = *pointer++;                                  // load data
        data = Swap(data);                                  // swap bytes for big endian
        length--;
        if(length == 0)
        {
         break;
        }
        *((unsigned short*)&CRCDAT) = data;                 // 16-bit word access to FIFO
    }

    CRCCONbits.CRCGO = 0;                                   // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                             // clear the interrupt flag
    *((unsigned short*)&CRCDAT) = data;                     // write last data into FIFO
    CRCCONbits.CRCGO = 1;                                   // resume CRC calculation
    while(!IFS0bits.CRCIF);                                 // wait until shifts are done
    crcResultCRCCCITT = (unsigned short)CRCWDAT;            // get CRC result (must be 0x9B4D)

    while(1);
    return 1;
}
```


Example 60-8: CRC-CCITT (16-Bit Polynomial with 16-Bit Data, Big-Endian, MOD Bit = 0)

```
// This macro is used to swap bytes for big endian
#define  Swap(x) __extension__({ \
unsigned long __x = (x), __v; \
__asm__  ("wsbh %0,%1;\n\t" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRCCCITT = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-CCITT

#define  CRCCCITT_POLYNOMIAL ((unsigned long)0x00001021)
#define  CRCCCITT_SEED_VALUE ((unsigned long)0x000084CF)    // non-direct initial value of 0xFFFF

    CRCCON = 0;
    CRCCONbits.MOD = 0;                                     // legacy mode
    CRCCONbits.ON = 1;                                      // enable CRC
    CRCCONbits.CRCISEL = 0;                                 // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 0;                                 // big endian
    CRCCONbits.DWIDTH = 16-1;                               // 16-bit data width
    CRCCONbits.PLEN = 16-1;                                 // 16-bit polynomial order
    CRCXOR = CRCCCITT_POLYNOMIAL;                           // set polynomial
    CRCWDAT = CRCCCITT_SEED_VALUE;                          // set initial value
    CRCCONbits.CRCGO = 1;                                   // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                           // wait if FIFO is full
        data = *pointer++;                                  // load data
        data = Swap(data);                                  // swap bytes for big endian
        *((unsigned short*)&CRCDAT) = data;                 // 16-bit word access to FIFO
    }

    while(CRCCONbits.CRCFUL);                               // wait if FIFO is full
    CRCCONbits.CRCGO = 0;                                   // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                             // clear the interrupt flag
    *((unsigned short*)&CRCDAT) = 0;                        // 16-bit dummy data to shift out CRC result
    CRCCONbits.CRCGO = 0;                                   // resume CRC calculation
    while(!IFS0bits.CRCIF);                                 // wait until shifts are done
    crcResultCRCCCITT = (unsigned short)CRCWDAT;            // get CRC result (must be 0x9B4D)

    while(1);
    return 1;
}
```

Example 60-9: CRC-32 (32-Bit Polynomial with 32-Bit Data, Little-Endian, MOD Bit = 1)

```
// ASCII bytes "12345678"
volatile unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
// function to reverse the bit order (OPTIONAL)
unsigned long ReverseBitOrder(unsigned long data);
volatile unsigned int crcResultCRC32 = 0;
int      main(void)
{
unsigned long*  pointer;
unsigned short  length;

// standard CRC-32

#define  CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define  CRC32_SEED_VALUE ((unsigned long)0xFFFFFFFF)   // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                 // alternate mode
    CRCCONbits.ON = 1;                                  // enable CRC
    CRCCONbits.CRCISEL = 0;                             // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                             // little endian
    CRCCONbits.DWIDTH = 32-1;                           // 32-bit data width
    CRCCONbits.PLEN = 32-1;                             // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                          // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                         // set initial value
    CRCCONbits.CRCGO = 1;                               // start CRC calculation
    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
        length--;
        if(length == 0)
        {
         break;
        }
        CRCDAT = *pointer++;                            // 32-bit word access to FIFO
    }
    CRCCONbits.CRCGO = 0;                               // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                         // clear the interrupt flag
    CRCDAT = *pointer;                                  // write last data into FIFO
    CRCCONbits.CRCGO = 1;                               // resume CRC calculation
    while(!IFS0bits.CRCIF);                             // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                           // get the final CRC result
    // OPTIONAL reverse CRC value bit order and invert (must be 0x9AE0DAAF)
    crcResultCRC32 = ~ReverseBitOrder(crcResultCRC32);
    while(1);
    return 1;
}
unsigned long ReverseBitOrder(unsigned long data)
{
unsigned long maskin;
unsigned long maskout;
unsigned long result = 0;
unsigned char i;
maskin = 0x80000000;
maskout = 0x00000001;
    for(i=0; i<32; i++)
    {
        if(data&maskin){
         result |= maskout;
        }
        maskin >>= 1;
        maskout <<= 1;
    }
return   result;
}
```


Example 60-10: CRC-32 (32-Bit Polynomial with 32-Bit Data, Little-Endian, MOD Bit = 0)

```
// ASCII bytes "12345678"
volatile unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
// function to reverse the bit order (OPTIONAL)
unsigned long ReverseBitOrder(unsigned long data);
volatile unsigned int crcResultCRC32 = 0;
int      main(void)
{
unsigned long*  pointer;
unsigned short  length;

// standard CRC-32

#define  CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define  CRC32_SEED_VALUE ((unsigned long)0x46AF6449)   // non-direct initial value of 0xFFFFFFFF

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                 // alternate mode
    CRCCONbits.ON = 1;                                  // enable CRC
    CRCCONbits.CRCISEL = 0;                             // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                             // little endian
    CRCCONbits.DWIDTH = 32-1;                           // 32-bit data width
    CRCCONbits.PLEN = 32-1;                             // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                          // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                         // set initial value
    CRCCONbits.CRCGO = 1;                               // start CRC calculation

    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
        CRCDAT = *pointer++;                            // 32-bit word access to FIFO
    }
    while(CRCCONbits.CRCFUL);                           // wait if FIFO is full
    CRCCONbits.CRCGO = 0;                               // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                         // clear the interrupt flag
    CRCDAT = 0;                                         // 32-bit dummy data to shift out the CRC result
    CRCCONbits.CRCGO = 1;                               // resume CRC calculation
    while(!IFS0bits.CRCIF);                             // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                           // get the final CRC result
    // OPTIONAL reverse CRC value bit order and invert (must be 0x9AE0DAAF)
    crcResultCRC32 = ~ReverseBitOrder(crcResultCRC32);
    while(1);
    return 1;
}

unsigned long ReverseBitOrder(unsigned long data)
{
unsigned long maskin;
unsigned long maskout;
unsigned long result = 0;
unsigned char i;
maskin = 0x80000000;
maskout = 0x00000001;
    for(i=0; i<32; i++)
    {
        if(data&maskin){
          result |= maskout;
        }
        maskin >>= 1;
        maskout <<= 1;
    }
return   result;
}
```

Example 60-11: Data Width Switching (32-Bit Polynomial, Little-Endian, MOD Bit = 1)

```
// ASCII bytes "12345678"
volatile unsigned long message1[] = {0x34333231,0x38373635};
// ASCII bytes "123"
volatile unsigned char message2[] = {'1','2','3'};
volatile unsigned long crcResultCRC32 = 0;
int      main(void)
{
unsigned char*  pointer8;
unsigned long* pointer32;
unsigned short  length;
#define   CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define   CRC32_SEED_VALUE ((unsigned long)0xFFFFFFFF)    // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                   // alternate mode
    CRCCONbits.ON = 1;                                    // enable CRC
    CRCCONbits.CRCISEL = 0;                               // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                               // little endian
    CRCCONbits.DWIDTH = 32-1;                             // 32-bit data width
    CRCCONbits.PLEN = 32-1;                               // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                            // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                           // set initial value
    CRCCONbits.CRCGO = 1;                                 // start CRC calculation

    pointer32 = (unsigned long*)message1;
    length = sizeof(message1)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                         // wait if FIFO is full
        length--;
        if(length == 0)
        {
         break;
        }
        CRCDAT = *pointer32++;                            // 32-bit word access to FIFO
    }
    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear the interrupt flag
    CRCDAT = *pointer32;                                  // write last 32-bit data into FIFO
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    CRCCONbits.DWIDTH = 8-1;                              // switch the data width to 8-bit

    pointer8 = (unsigned char*)message2;                  // calculate CRC
    length = sizeof(message2)/sizeof(unsigned char);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                         // wait if FIFO is full
        length--;
        if(length == 0)
        {
         break;
        }
        *((unsigned char*)&CRCDAT) = *pointer8++;         // byte access to FIFO
    }
    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear the interrupt flag
    *((unsigned char*)&CRCDAT) = *pointer8;               // write last 8-bit data into FIFO
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                             // get the final CRC result (must be 0xE092727E)

    while(1);
    return 1;
}
```


Example 60-12: Data Width Switching (32-Bit Polynomial, Little-Endian, MOD Bit = 0)

```
// ASCII bytes "12345678"
volatile  unsigned long message1[] = {0x34333231,0x38373635};
// ASCII bytes "123"
volatile  unsigned char message2[] = {'1','2','3'};
volatile  unsigned long crcResultCRC32 = 0;
int       main(void)
{
unsigned  char*  pointer8;
unsigned  long* pointer32;
unsigned  short  length;
#define   CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define   CRC32_SEED_VALUE ((unsigned long)0x46AF6449)    // non-direct initial value of 0xFFFFFFFF

    CRCCON = 0;
    CRCCONbits.MOD = 0;                                   // alternate mode
    CRCCONbits.ON = 1;                                    // enable CRC
    CRCCONbits.CRCISEL = 0;                               // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                               // little endian
    CRCCONbits.DWIDTH = 32-1;                             // 32-bit data width
    CRCCONbits.PLEN = 32-1;                               // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                            // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                           // set initial value
    CRCCONbits.CRCGO = 1;                                 // start CRC calculation

    pointer32 = (unsigned long*)message1;
    length = sizeof(message1)/sizeof(unsigned long)-1;
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                         // wait if FIFO is full
        CRCDAT = *pointer32++;                            // 32-bit word access to FIFO
    }
    while(CRCCONbits.CRCFUL);                             // wait if FIFO is full
    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear interrupt flag
    CRCDAT = *pointer32;                                  // write last 32-bit data into FIFO
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done

    CRCCONbits.DWIDTH = 8-1;                              // switch the data width to 8-bit
    pointer8 = (unsigned char*)message2;                  // calculate CRC
    length = sizeof(message2)/sizeof(unsigned char)-1;
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                         // wait if FIFO is full
        *((unsigned char*)&CRCDAT) = *pointer8++;         // byte access to FIFO
    }
    while(CRCCONbits.CRCFUL);                             // wait if FIFO is full
    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear interrupt flag
    *((unsigned char*)&CRCDAT) = *pointer8;               // write last 8-bit data into FIFO
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    CRCCONbits.DWIDTH = 32-1;                             // switch data width to polynomial length 32-bit

    CRCCONbits.CRCGO = 0;                                 // suspend CRC calculation
    IFS0CLR = _IFS0_CRCIF_MASK;                           // clear interrupt flag
    CRCDAT = 0;                                           // write dummy data to shift out the CRC result
    CRCCONbits.CRCGO = 1;                                 // resume CRC calculation
    while(!IFS0bits.CRCIF);                               // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                             // get the final CRC result (must be 0xE092727E)

    while(1);
    return 1;
}
```

60.6.4 Application Examples for CRC Modules with Persistent Interrupt

Example 60-13 through Example 60-22 show typical code for different combinations of polynomial length, data width, shift direction and CRC Engine modes for modules with persistent interrupt.

Example 60-13: CRC-SMBus (8-Bit Polynomial with 32-Bit Data, Big-Endian, MOD Bit = 1)

```
// This macro is used to swap bytes for big endian
#define  Swap(x) __extension__({ \
unsigned long __x = (x), __v; \
__asm__  ("wsbh %0,%1;\n\t" \
"rotr %0,16" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})

// ASCII bytes "12345678"
volatile unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
volatile unsigned char crcResultCRCSMBUS = 0;
int      main (void)
{
unsigned long*  pointer;
unsigned short length;
unsigned long  data;

// standard CRC-SMBUS

#define  CRCSMBUS_POLYNOMIAL ((unsigned long)0x00000007)
#define  CRCSMBUS_SEED_VALUE ((unsigned long)0x00000000) // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                  // alternate mode
    CRCCONbits.ON = 1;                                   // enable CRC
    CRCCONbits.LENDIAN = 0;                              // big endian
    CRCCONbits.CRCISEL = 0;                              // interrupt when all shifts are done
    CRCCONbits.DWIDTH = 32-1;                            // 32-bit data width
    CRCCONbits.PLEN = 8-1;                               // 8-bit polynomial order
    CRCXOR = CRCSMBUS_POLYNOMIAL;                        // set polynomial
    CRCWDAT = CRCSMBUS_SEED_VALUE;                       // set initial value
    CRCCONbits.CRCGO = 1;                                // start CRC calculation

    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                        // wait if FIFO is full
        data = *pointer++;                               // load from little endian
        data = Swap(data);                               // swap bytes for big endian
        length--;
        if(length == 0)
        {
          break;
        }
        CRCDAT = data;                                   // 32-bit word access to FIFO
    }
    CRCDAT = data;                                       // write last data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                          // clear the interrupt flag

    while(!IFS0bits.CRCIF);                              // wait until shifts are done
    crcResultCRCSMBUS = (unsigned char)CRCWDAT&0x00ff;   // get CRC result (must be 0xC7)

    while(1);
    return 1;
}
```

Example 60-14: CRC-SMBus (8-Bit Polynomial with 32-Bit Data, Little-Endian, MOD Bit = 0)

```
// This macro is used to swap bytes for big endian
#define  Swap(x) __extension__({ \
unsigned long __x = (x), __v; \
__asm__  ("wsbh %0,%1;\n\t" \
"rotr %0,16" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})

// ASCII bytes "12345678"
volatile unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
volatile unsigned char crcResultCRCSMBUS = 0;
int      main (void)
{
unsigned long* pointer;
unsigned short length;
unsigned long  data;

// standard CRC-SMBUS

#define  CRCSMBUS_POLYNOMIAL ((unsigned long)0x00000007)
#define  CRCSMBUS_SEED_VALUE ((unsigned long)0x00000000) // non-direct initial value of zero

    CRCCON = 0;
    CRCCONbits.MOD = 0;                                 // legacy mode
    CRCCONbits.ON = 1;                                  // enable CRC
    CRCCONbits.LENDIAN = 0;                             // big endian
    CRCCONbits.CRCISEL = 0;                             // interrupt when all shifts are done
    CRCCONbits.DWIDTH = 32-1;                           // 32-bit data width
    CRCCONbits.PLEN = 8-1;                              // 8-bit polynomial order
    CRCXOR = CRCSMBUS_POLYNOMIAL;                       // set polynomial
    CRCWDAT = CRCSMBUS_SEED_VALUE;                      // set initial value
    CRCCONbits.CRCGO = 1;                               // start CRC calculation
    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
        data = *pointer++;                              // load from little endian
        data = Swap(data);                              // swap bytes for big endian
        length--;
        if(length == 0)
        {
         break;
        }
        CRCDAT = data;                                  // 32-bit word access to FIFO
    }

    CRCDAT = data;                                      // write last data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                         // clear the interrupt flag
    while(!IFS0bits.CRCIF);                             // wait until shifts are done
    CRCCONbits.DWIDTH = 8-1;                            // switch data width to polynomial length 8-bit
    *((unsigned char*)&CRCDAT) = 0;                     // 8-bit dummy data to shift out the CRC result
    IFS0CLR = _IFS0_CRCIF_MASK;                         // clear the interrupt flag
    while(!IFS0bits.CRCIF);                             // wait until shifts are done
    crcResultCRCSMBUS = (unsigned char)CRCWDAT&0x00ff;  // get CRC result (must be 0xC7)

    while(1);
    return 1;
}
```

Example 60-15: CRC-16 (16-Bit Data with 32-Bit Polynomial, Little-Endian, MOD Bit = 1)

```
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRC16 = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-16

#define  CRC16_POLYNOMIAL ((unsigned long)0x00008005)
#define  CRC16_SEED_VALUE ((unsigned long)0x00000000) // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                              // alternate mode
    CRCCONbits.ON = 1;                               // enable CRC
    CRCCONbits.CRCISEL = 0;                          // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                          // little endian
    CRCCONbits.DWIDTH = 16-1;                        // 16-bit data width
    CRCCONbits.PLEN = 16-1;                          // 16-bit polynomial order
    CRCXOR = CRC16_POLYNOMIAL;                       // set polynomial
    CRCWDAT = CRC16_SEED_VALUE;                      // set initial value
    CRCCONbits.CRCGO = 1;                            // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                    // wait if FIFO is full
        data = *pointer++;                           // load data
        length--;
        if(length == 0)
        {
          break;
        }
        *((unsigned short*)&CRCDAT) = data;          // 16-bit word access to FIFO
    }

    *((unsigned short*)&CRCDAT) = data;              // write last data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                      // clear the interrupt flag
    while(!IFS0bits.CRCIF);                          // wait until shifts are done
    crcResultCRC16 = (unsigned short)CRCWDAT;        // get CRC result (must be 0xE716)

    while(1);
    return 1;
}
```

Example 60-16: CRC-16 (16-Bit Data, 16-Bit Polynomial, Little-Endian, MOD Bit = 0)

```
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRC16 = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-16

#define  CRC16_POLYNOMIAL ((unsigned long)0x00008005)
#define  CRC16_SEED_VALUE ((unsigned long)0x00000000) // non-direct initial value of zero

    CRCCON = 0;
    CRCCONbits.MOD = 0;                          // legacy mode
    CRCCONbits.ON = 1;                           // enable CRC
    CRCCONbits.CRCISEL = 0;                      // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                      // little endian
    CRCCONbits.DWIDTH = 16-1;                    // 16-bit data width
    CRCCONbits.PLEN = 16-1;                      // 16-bit polynomial order
    CRCXOR = CRC16_POLYNOMIAL;                   // set polynomial
    CRCWDAT = CRC16_SEED_VALUE;                  // set initial value
    CRCCONbits.CRCGO = 1;                        // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                // wait if FIFO is full
        data = *pointer++;                       // load data
        *((unsigned short*)&CRCDAT) = data;      // 16-bit word access to FIFO
    }

    while(CRCCONbits.CRCFUL);                    // wait if FIFO is full
    *((unsigned short*)&CRCDAT) = 0;             // 16-bit dummy data to shift out CRC result
    IFS0CLR = _IFS0_CRCIF_MASK;                  // clear the interrupt flag
    while(!IFS0bits.CRCIF);                      // wait until shifts are done
    crcResultCRC16 = (unsigned short)CRCWDAT;    // get CRC result (must be 0xE716)

    while(1);
    return 1;
}
```

Example 60-17: CRC-CCITT (16-Bit Polynomial with 16-Bit Data, Big-Endian, MOD Bit = 1)

```
// This macro is used to swap bytes for big endian
#define  Swap(x) __extension__({ \
unsigned long __x = (x), __v; \
__asm__  ("wsbh %0,%1;\n\t" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRCCCITT = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-CCITT

#define  CRCCCITT_POLYNOMIAL ((unsigned long)0x00001021)
#define  CRCCCITT_SEED_VALUE ((unsigned long)0x0000FFFF) // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                 // alternate mode
    CRCCONbits.ON = 1;                                  // enable CRC
    CRCCONbits.CRCISEL = 0;                             // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 0;                             // big endian
    CRCCONbits.DWIDTH = 16-1;                           // 16-bit data width
    CRCCONbits.PLEN = 16-1;                             // 16-bit polynomial order
    CRCXOR = CRCCCITT_POLYNOMIAL;                       // set polynomial
    CRCWDAT = CRCCCITT_SEED_VALUE;                      // set initial value
    CRCCONbits.CRCGO = 1;                               // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
        data = *pointer++;                              // load data
        data = Swap(data);                              // swap bytes for big endian
        length--;
        if(length == 0)
        {
         break;
        }
        *((unsigned short*)&CRCDAT) = data;             // 16-bit word access to FIFO
    }

    *((unsigned short*)&CRCDAT) = data;                 // write last data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                         // clear the interrupt flag
    while(!IFS0bits.CRCIF);                             // wait until shifts are done
    crcResultCRCCCITT = (unsigned short)CRCWDAT;        // get CRC result (must be 0x9B4D)

    while(1);
    return 1;
}
```


Example 60-18: CRC-CCITT (16-Bit Polynomial with 16-Bit Data, Big-Endian, MOD Bit = 0)

```
// This macro is used to swap bytes for big endian
#define  Swap(x) __extension__({ \
unsigned long __x = (x), __v; \
__asm__  ("wsbh %0,%1;\n\t" \
: "=d" (__v) \
: "d" (__x)); \
__v; \
})
// ASCII bytes "87654321"
volatile unsigned short message[] = {0x3738,0x3536,0x3334,0x3132};
volatile unsigned short crcResultCRCCCITT = 0;
int      main (void)
{
unsigned short* pointer;
unsigned short  length;
unsigned short  data;

// standard CRC-CCITT

#define  CRCCCITT_POLYNOMIAL ((unsigned long)0x00001021)
#define  CRCCCITT_SEED_VALUE ((unsigned long)0x000084CF)// non-direct initial value of 0xFFFF

    CRCCON = 0;
    CRCCONbits.MOD = 0;                                 // legacy mode
    CRCCONbits.ON = 1;                                  // enable CRC
    CRCCONbits.CRCISEL = 0;                             // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 0;                             // big endian
    CRCCONbits.DWIDTH = 16-1;                           // 16-bit data width
    CRCCONbits.PLEN = 16-1;                             // 16-bit polynomial order
    CRCXOR = CRCCCITT_POLYNOMIAL;                       // set polynomial
    CRCWDAT = CRCCCITT_SEED_VALUE;                      // set initial value
    CRCCONbits.CRCGO = 1;                               // start CRC calculation

    pointer = (unsigned short*)message;
    length = sizeof(message)/sizeof(unsigned short);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
        data = *pointer++;                              // load data
        data = Swap(data);                              // swap bytes for big endian
        *((unsigned short*)&CRCDAT) = data;             // 16-bit word access to FIFO
    }

    while(CRCCONbits.CRCFUL);                           // wait if FIFO is full
    *((unsigned short*)&CRCDAT) = 0;                    // 16-bit dummy data to shift out CRC result
    IFS0CLR = _IFS0_CRCIF_MASK;                         // clear the interrupt flag
    while(!IFS0bits.CRCIF);                             // wait until shifts are done
    crcResultCRCCCITT = (unsigned short)CRCWDAT;        // get CRC result (must be 0x9B4D)

    while(1);
    return 1;
}
```

Example 60-19: CRC-32 (32-Bit Polynomial with 32-Bit Data, Little-Endian, MOD Bit = 1)

```
// ASCII bytes "12345678"
volatile unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
// function to reverse the bit order (OPTIONAL)
unsigned long ReverseBitOrder(unsigned long data);
volatile unsigned int crcResultCRC32 = 0;
int      main(void)
{
unsigned long*  pointer;
unsigned short  length;

// standard CRC-32

#define  CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define  CRC32_SEED_VALUE ((unsigned long)0xFFFFFFFF)    // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                                  // alternate mode
    CRCCONbits.ON = 1;                                   // enable CRC
    CRCCONbits.CRCISEL = 0;                              // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                              // little endian
    CRCCONbits.DWIDTH = 32-1;                            // 32-bit data width
    CRCCONbits.PLEN = 32-1;                              // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                           // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                          // set initial value
    CRCCONbits.CRCGO = 1;                                // start CRC calculation
    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                        // wait if FIFO is full
        length--;
        if(length == 0)
        {
         break;
        }
        CRCDAT = *pointer++;                             // 32-bit word access to FIFO
    }
    CRCDAT = *pointer;                                   // write last data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                          // clear the interrupt flag
    while(!IFS0bits.CRCIF);                              // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                            // get the final CRC result
    // OPTIONAL reverse CRC value bit order and invert (must be 0x9AE0DAAF)
    crcResultCRC32 = ~ReverseBitOrder(crcResultCRC32);
    while(1);
    return 1;
}
unsigned long ReverseBitOrder(unsigned long data)
{
unsigned long maskin;
unsigned long maskout;
unsigned long result = 0;
unsigned char i;
maskin = 0x80000000;
maskout = 0x00000001;
    for(i=0; i<32; i++)
    {
        if(data&maskin){
            result |= maskout;
        }
        maskin >>= 1;
        maskout <<= 1;
    }
return   result;
}
```

Example 60-20: CRC-32 (32-Bit Polynomial with 32-Bit Data, Little-Endian, MOD Bit = 0)

```
// ASCII bytes "12345678"
volatile unsigned char __attribute__((aligned(4))) message[] = {'1','2','3','4','5','6','7','8'};
// function to reverse the bit order (OPTIONAL)
unsigned long ReverseBitOrder(unsigned long data);
volatile unsigned int crcResultCRC32 = 0;
int      main(void)
{
unsigned long*  pointer;
unsigned short  length;

// standard CRC-32

#define  CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define  CRC32_SEED_VALUE ((unsigned long)0x46AF6449) // non-direct initial value of 0xFFFFFFFF

    CRCCON = 0;
    CRCCONbits.MOD = 1;                          // alternate mode
    CRCCONbits.ON = 1;                           // enable CRC
    CRCCONbits.CRCISEL = 0;                      // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                      // little endian
    CRCCONbits.DWIDTH = 32-1;                    // 32-bit data width
    CRCCONbits.PLEN = 32-1;                      // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                   // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                  // set initial value
    CRCCONbits.CRCGO = 1;                        // start CRC calculation

    pointer = (unsigned long*)message;
    length = sizeof(message)/sizeof(unsigned long);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                // wait if FIFO is full
        CRCDAT = *pointer++;                     // 32-bit word access to FIFO
    }
    while(CRCCONbits.CRCFUL);                    // wait if FIFO is full
    CRCDAT = 0;                                  // 32-bit dummy data to shift out the CRC result
    IFS0CLR = _IFS0_CRCIF_MASK;                  // clear the interrupt flag
    while(!IFS0bits.CRCIF);                      // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                    // get the final CRC result
    // OPTIONAL reverse CRC value bit order and invert (must be 0x9AE0DAAF)
    crcResultCRC32 = ~ReverseBitOrder(crcResultCRC32);
    while(1);
    return 1;
}

unsigned long ReverseBitOrder(unsigned long data)
{
unsigned long maskin;
unsigned long maskout;
unsigned long result = 0;
unsigned char i;
maskin = 0x80000000;
maskout = 0x00000001;
    for(i=0; i<32; i++)
    {
        if(data&maskin){
               result |= maskout;
        }
        maskin >>= 1;
        maskout <<= 1;
    }
return   result;
}
```

Example 60-21: Data Width Switching (32-Bit Polynomial, Little-Endian, MOD Bit = 1)

```
// ASCII bytes "12345678"
volatile unsigned long message1[] = {0x34333231,0x38373635};
// ASCII bytes "123"
volatile unsigned char message2[] = {'1','2','3'};
volatile unsigned long crcResultCRC32 = 0;
int      main(void)
{
unsigned char*  pointer8;
unsigned long* pointer32;
unsigned short  length;
#define   CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define   CRC32_SEED_VALUE ((unsigned long)0xFFFFFFFF) // direct initial value

    CRCCON = 0;
    CRCCONbits.MOD = 1;                             // alternate mode
    CRCCONbits.ON = 1;                              // enable CRC
    CRCCONbits.CRCISEL = 0;                         // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                         // little endian
    CRCCONbits.DWIDTH = 32-1;                       // 32-bit data width
    CRCCONbits.PLEN = 32-1;                         // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                      // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                     // set initial value
    CRCCONbits.CRCGO = 1;                           // start CRC calculation

    pointer32 = (unsigned long*)message1;
    length = sizeof(message1)/sizeof(unsigned long);
    while(1)
    {
        while(CRCCONbits.CRCFUL);                   // wait if FIFO is full
        length--;
        if(length == 0)
        {
          break;
        }
        CRCDAT = *pointer32++;                      // 32-bit word access to FIFO
    }
    CRCDAT = *pointer32;                            // write last 32-bit data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                     // clear the interrupt flag
    while(!IFS0bits.CRCIF);                         // wait until shifts are done
    CRCCONbits.DWIDTH = 8-1;                        // switch the data width to 8-bit

    pointer8 = (unsigned char*)message2;            // calculate CRC
    length = sizeof(message2)/sizeof(unsigned char);
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                   // wait if FIFO is full
        length--;
        if(length == 0)
        {
          break;
        }
        *((unsigned char*)&CRCDAT) = *pointer8++;   // byte access to FIFO
    }
    *((unsigned char*)&CRCDAT) = *pointer8;         // write last 8-bit data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                     // clear the interrupt flag
    while(!IFS0bits.CRCIF);                         // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                       // get the final CRC result (must be 0xE092727E)

    while(1);
    return 1;
}
```


Example 60-22: Data Width Switching (32-Bit Polynomial, Little-Endian, MOD Bit = 0)

```
// ASCII bytes "12345678"
volatile unsigned long message1[] = {0x34333231,0x38373635};
// ASCII bytes "123"
volatile unsigned char message2[] = {'1','2','3'};
volatile unsigned long crcResultCRC32 = 0;
int      main(void)
{
unsigned char*  pointer8;
unsigned long* pointer32;
unsigned short  length;
#define  CRC32_POLYNOMIAL ((unsigned long)0x04C11DB7)
#define  CRC32_SEED_VALUE ((unsigned long)0x46AF6449)  // non-direct initial value of 0xFFFFFFFF

    CRCCON = 0;
    CRCCONbits.MOD = 0;                             // alternate mode
    CRCCONbits.ON = 1;                              // enable CRC
    CRCCONbits.CRCISEL = 0;                         // interrupt when all shifts are done
    CRCCONbits.LENDIAN = 1;                         // little endian
    CRCCONbits.DWIDTH = 32-1;                       // 32-bit data width
    CRCCONbits.PLEN = 32-1;                         // 32-bit polynomial order
    CRCXOR = CRC32_POLYNOMIAL;                      // set polynomial
    CRCWDAT = CRC32_SEED_VALUE;                     // set initial value
    CRCCONbits.CRCGO = 1;                           // start CRC calculation

    pointer32 = (unsigned long*)message1;
    length = sizeof(message1)/sizeof(unsigned long)-1;
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                   // wait if FIFO is full
        CRCDAT = *pointer32++;                      // 32-bit word access to FIFO
    }
    while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
    CRCDAT = *pointer32;                            // write last 32-bit data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                     // clear interrupt flag
    while(!IFS0bits.CRCIF);                         // wait until shifts are done

    CRCCONbits.DWIDTH = 8-1;                        // switch the data width to 8-bit
    pointer8 = (unsigned char*)message2;            // calculate CRC
    length = sizeof(message2)/sizeof(unsigned char)-1;
    while(length--)
    {
        while(CRCCONbits.CRCFUL);                   // wait if FIFO is full
        *((unsigned char*)&CRCDAT) = *pointer8++;   // byte access to FIFO
    }
    while(CRCCONbits.CRCFUL);                       // wait if FIFO is full
    *((unsigned char*)&CRCDAT) = *pointer8;         // write last 8-bit data into FIFO
    IFS0CLR = _IFS0_CRCIF_MASK;                     // clear interrupt flag
    while(!IFS0bits.CRCIF);                         // wait until shifts are done
    CRCCONbits.DWIDTH = 32-1;                       // switch data width to polynomial length 32-bit

    CRCDAT = 0;                                     // write dummy data to shift out the CRC result
    IFS0CLR = _IFS0_CRCIF_MASK;                     // clear interrupt flag
    while(!IFS0bits.CRCIF);                         // wait until shifts are done
    crcResultCRC32 = CRCWDAT;                       // get the final CRC result (must be 0xE092727E)

    while(1);
    return 1;
}
```

60.7 OPERATION IN POWER SAVE MODES

60.7.1 Sleep Mode

If Sleep mode is entered while the module is operating, the module is suspended in its current state until clock execution resumes.

60.7.2 Idle Mode

To continue full module operation in Idle mode, the SIDL bit must be cleared prior to entry into the mode.

If SIDL = 1, the module behaves the same way as it does in Sleep mode; pending interrupt events will be passed on, even though the module clocks are not available.

60.8 EFFECTS OF A RESET

Following a device Reset, all of the CRC Control registers (CRCCON, CRCXOR and CRCWDAT) are reset to '0x00000000'. This disables the CRC module. Any calculation that was in progress will terminate.

60.9 RELATED APPLICATION NOTES

This section lists application notes that are related to this section of the manual. These application notes may not be written specifically for the PIC32 device family, but the concepts are pertinent and could be used with modification and possible limitations. The current application notes related to the 32-Bit Programmable Cyclic Redundancy Check (CRC) are:

Title	Application Note #
No related application notes at this time.	

Note: Please visit the Microchip web site (www.microchip.com) for additional application notes and code examples for the PIC32 family of devices.

60.10 REVISION HISTORY

Revision A (May 2015)

This is the initial released revision of this document.

Revision B (March 2016)

- Figures:
  - Updates Figure 60-1.
- Examples:
  - Updates Example 60-2 and Example 60-3 through Example 60-12.

Revision C (June 2016)

Updates **Section 60.5.4.4 “CRC Result”** and **Section 60.5.5 “Interrupt Operation”**.

Adds **Section 60.6.4 “Application Examples for CRC Modules with Persistent Interrupt”**.

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949 ═

Trademarks

ISBN: 978-1-5224-0688-4

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110

Canada - Toronto
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon

Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Dongguan
Tel: 86-769-8702-9880

China - Hangzhou
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

ASIA/PACIFIC

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-3019-1500

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7828

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Dusseldorf
Tel: 49-2129-3766400

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Venice
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Poland - Warsaw
Tel: 48-22-3325737

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

07/14/15