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 dsPIC33/PIC24 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

1.0 INTRODUCTION

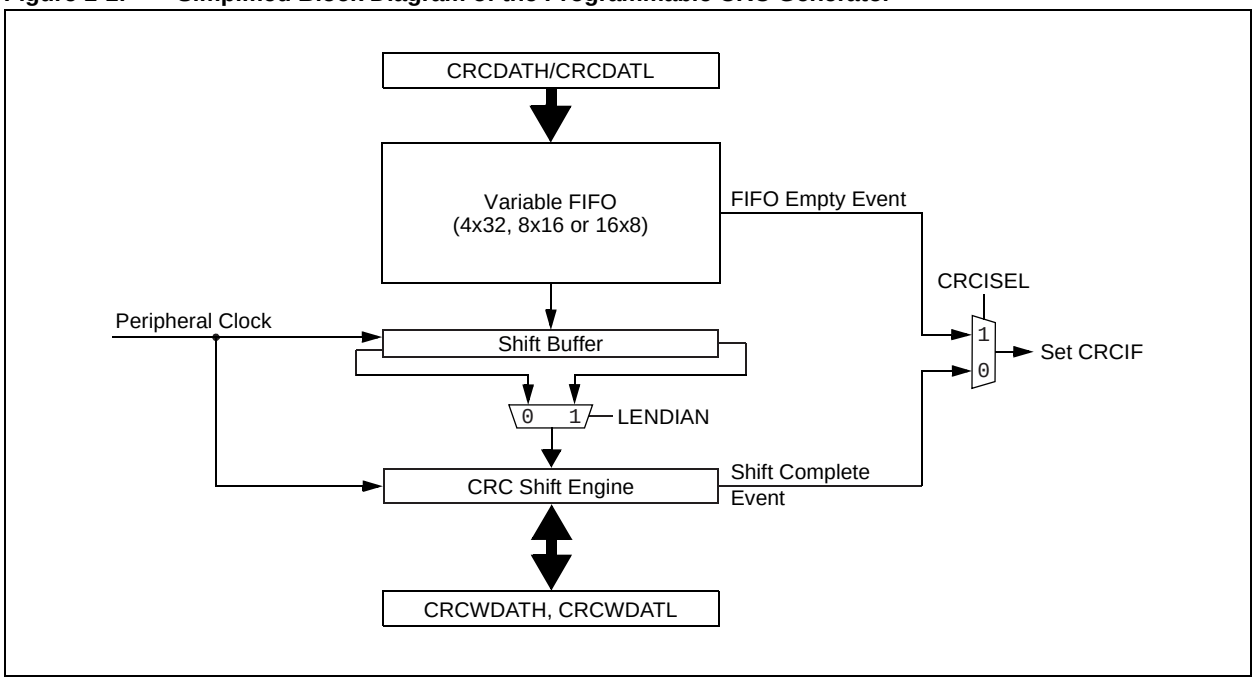
The 32-Bit Programmable Cyclic Redundancy Check (CRC) module is a software-configurable CRC generator. The module provides a hardware implemented method of quickly generating checksums for various communication and security applications. The CRC engine calculates the CRC checksum without CPU intervention; 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 1-1.

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



2.0 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 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. Like in 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 2-1:

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

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 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.

3.0 CRC REGISTERS

Different registers associated with the CRC module are described in detail in this section. There are eight registers in this module. These are mapped to the data RAM space as Special Function Registers (SFRs) in dsPIC33/PIC24 devices:

- **CRCCON1: CRC Control Register 1**
- **CRCCON2: CRC Control Register 2**
- **CRCXORL: CRC XOR Low Register**
- **CRCXORH: CRC XOR High Register**
- **CRCDATL: CRC Data Low Register**
- **CRCDATH: CRC Data High Register**
- **CRCWDATL: CRC Shift Low Register**
- **CRCWDATH: CRC Shift High Register**

The CRCCON1 (Register 3-1) and CRCCON2 (Register 3-2) registers control the operation of the module, and configure various settings. The CRCXORL/H registers (Register 3-3 and Register 3-4) select the polynomial terms to be used in the CRC equation. The CRCDATL/H and CRCWDATL/H registers are each register pairs that serve as buffers for the double-word input data and CRC processed output, respectively.

3.1 Register Maps

A summary of the Special Function Registers associated with the dsPIC33/PIC24 32-Bit Programmable Cyclic Redundancy Check (CRC) module is provided in Table 3-1.The corresponding registers appear after the summaries, followed by a detailed description of each register.

Table 3-1: Special Function Registers Associated with the Programmable CRC Module(1)

File Name	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
CRCCON1	CRCEN	—	CSIDL	VWORD<4:0>					CRCFUL	CRCMPT	CRCISEL	CRCGO	LENDIAN	MOD(2)	—	—	0040
CRCCON2	—	—	—	DWIDTH<4:0>					—	—	—	PLEN<4:0>					0000
CRCXORL	X<15:1>															—	0000
CRCXORH	X<31:16>																0000
CRCDATL	DATA<15:0>																0000
CRCDATH	DATA<31:16>																0000
CRCWDATL	SDATA<15:0>																0000
CRCWDATH	SDATA<31:16>																0000

Legend: — = unimplemented, read as '0'.

Note 1: Refer to the specific device data sheet for memory map details.

2: This bit is not available on all devices. Refer to the specific device data sheet for details.

Register 3-1: CRCCON1: CRC Control Register 1

R/W-0	U-0	R/W-0	R-0	R-0	R-0	R-0	R-0
CRCEN	—	CSIDL	VWORD<4:0>				
bit 15							bit 8

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(1)	—	—
bit 7							bit 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 15 **CRCEN:** CRC Enable bit
1 = Enables module
0 = Disables module

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

bit 13 **CSIDL:** CRC Stop in Idle Mode bit
1 = Discontinues module operation when device enters Idle mode
0 = Continues module operation in Idle mode

bit 12-8 **VWORD<4:0>:** Counter Value bits
Indicates the number of valid words in the FIFO. Has a maximum value of 16 when DWIDTH<4:0> ≤ 7 (data words, 8-bit wide or less). Has a maximum value of 8 when DWIDTH<4:0> ≤ 15 (data words from 9 to 16-bit wide). Has a maximum value of 4 when DWIDTH<4:0> ≤ 31 (data words from 17 to 32-bit wide).

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 = Interrupt on FIFO empty; final word of data is still shifted through CRC
0 = Interrupt on shift complete (FIFO is empty and no data is shifted from the shift buffer)

bit 4 **CRCGO:** Start CRC 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)
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'

Note 1: This bit is not available on all devices. Refer to the specific device data sheet for details.

Register 3-2: CRCCON2: CRC Control Register 2

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	DWIDTH<4:0>				
bit 15							bit 8

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	PLEN<4:0>				
bit 7							bit 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 15-13 **Unimplemented:** Read as ‘0’

bit 12-8 **DWIDTH<4:0>:** Data Word Width Configuration bits
Configures the width of the data word (Data Word Width – 1).

bit 7-5 **Unimplemented:** Read as ‘0’

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

Register 3-3: CRCXORL: CRC XOR Low Register

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>							
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0
X<7:1>							—
bit 7							bit 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 15-1 **X<15:1>:** XOR of Polynomial Term x^n Enable bits

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

Register 3-4: CRCXORH: CRC XOR High Register

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>							
bit 15							bit 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<23:16>							
bit 7							bit 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 15-0 **X<31:16>:** XOR of Polynomial Term x^n Enable bits

Register 3-5: CRCDATL: CRC Data Low Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DATA<15:8>							
bit 15							bit 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
DATA<7:0>							
bit 7							bit 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 15-0 **DATA<15:0>:** CRC Input Data bits
Writing to this register fills the FIFO; reading from this register returns ‘0’.

Register 3-6: CRCDATH: CRC Data High Register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
DATA<31:24>							
bit 15							bit 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
DATA<23:16>							
bit 7							bit 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 15-0 **DATA<31:16>:** CRC Input Data bits
Writing to this register fills the FIFO; reading from this register returns ‘0’.

Register 3-7: CRCWDATL: CRC Shift Low Register

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>							
bit 15							bit 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<7:0>							
bit 7							bit 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 15-0 **SDATA<15:0>:** CRC Shift Register bits

Writing to this register writes to the CRC Shift register through the CRC write bus. Reading from this register reads the CRC read bus.

Register 3-8: CRCWDATH: CRC Shift High Register

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>							
bit 15							bit 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<23:16>							
bit 7							bit 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 15-0 **DATA<31:16>:** CRC Shift Register bits

Writing to this register writes to the CRC Shift register through the CRC write bus. Reading from this register reads the CRC read bus.

4.0 CRC ENGINE

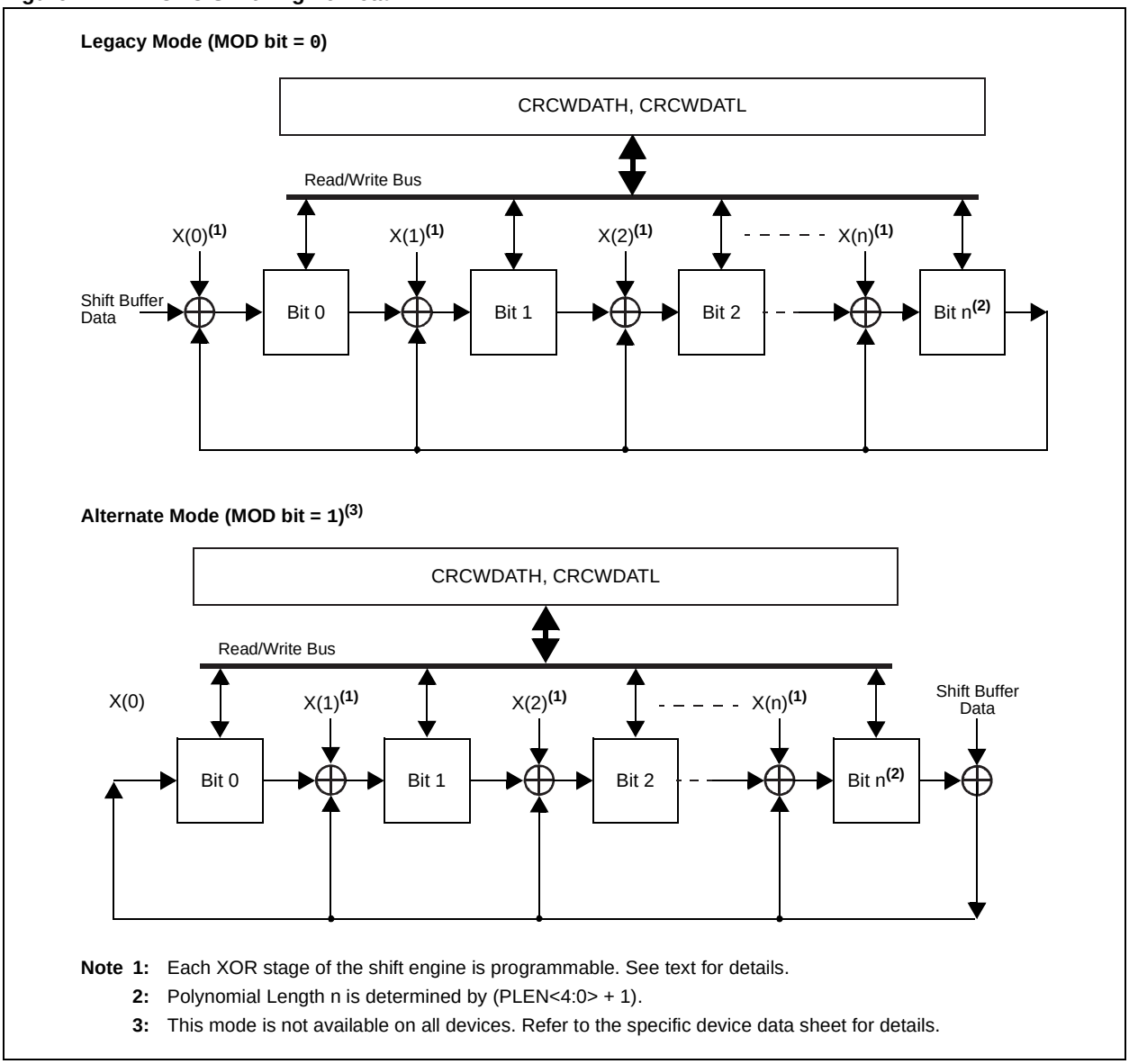
4.1 Generic CRC Engine

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 (CRCCON1<2>). A simplified diagram of the CRC shift engine is shown in Figure 4-1.

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 registers. 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 CRCCON2 register (CRCCON2<4:0>). 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 registers.

Figure 4-1: 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).

3: This mode is not available on all devices. Refer to the specific device data sheet for details.

5.0 CONTROL LOGIC

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 (CRCCON2<4:0>). The CRCXOR registers control 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 5-1).To program these polynomials into the CRC generator, set the register bits as shown in Table 5-1.

Equation 5-1:

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

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 5-1: 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.

5.2 Data Shift Direction

The LENDIAN bit (CRCCON1<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 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.

dsPIC33/PIC24 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 registers).

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 (CRCCON2<12:8>). It can be configured to any value, between 1 and 32 bits. The logic associated with the FIFO contains a 5-bit counter, VWORD<4:0> bits (CRCCON1<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 registers. Reading the CRCDAT registers 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 registers 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 eight bits or less, the FIFO increments on a write to the lower byte of the CRCDATL register (a byte access to the CRCDATL register must be used). The smallest data element that can be written into the FIFO is one byte.

For example, if DWIDTH<4:0> is five, then the size of the data is DWIDTH<4:0> + 1 or six. The data is written as a whole byte; the two unused upper bits are ignored. Once the data byte is written into the CRCDATL register, the value of the VWORD<4:0> bits (CRCCON1<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 CRCDATL register (16-bit word access to the CRCDATL 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 CRCDATL register.

When the data width is greater than 16 bits, any write to the CRCDATH register increments the VWORD<4:0> bits by one. Writing the lower word into the CRCDATL register must be done before writing the upper word into the CRCDATH register. Unused upper data bits are ignored.

5.4 CRC Engine Interface

5.4.1 FIFO TO CRC SHIFT ENGINE

To start moving the data from the FIFO to the CRC shift buffer, the CRCGO bit (CRCCON1<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 registers, 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 the CRCEN bit is '0'.

5.4.2 NUMBER OF CLOCK CYCLES TO SHIFT DATA

The data from FIFO goes to the shift buffer. It takes two peripheral clock cycles to start moving the data words from 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 six bits (DWIDTH<4:0> + 1) and six cycles are required to shift the data. In this case, only six 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 four bits of each word are ignored.

5.4.3 CRC INITIAL VALUE

The access to the CRC shift engine is provided through the CRCWDAT registers. These registers 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 (CRCCON1<2>).

In Alternate mode (MOD bit = 1, not available on all devices), the CRC initial value must be in direct form.

In Legacy mode (MOD bit = 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 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 registers (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 registers, 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 registers 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 5-1 shows a possible software routine to get the non-direct initial value from the direct initial value.

Example 5-1: 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 (CRCEN = 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 registers.
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 registers.
8. Write a dummy data to the CRCDAT registers and wait two 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 registers and wait for the CRC interrupt flag to set.
9. Read the value from the CRCWDAT registers.
10. Reverse the bit order of the read result; it will give the final non-direct initial value.

Example 5-2 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 registers again. In this case, the CRC direct initial value will be an intermediate CRC result read.

Example 5-2: 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)
{
   CRCCON1 = 0;
   CRCCON2 = 0;

   CRCCON1bits.CRCEN   = 1;                                  // enable CRC
   CRCCON1bits.CRCISEL = 0;                                  // interrupt when all shifts are done
   CRCCON2bits.DWIDTH  = polynomialOrder-1;                  // data width
   CRCCON2bits.PLEN    = polynomialOrder-1;                  // polynomial length
   CRCCON1bits.CRCGO   = 1;                                  // start CRC calculation


   polynomial >>= 1;                                         // shift the polynomial right

   polynomial = ReverseBitOrder(polynomial, polynomialOrder); // reverse bits order of the
                                                             // polynomial
   CRCXORL  = (unsigned short)(polynomial&0x0000FFFF);       // set the reversed polynomial
   CRCXORH  = (unsigned short)(polynomial>>16);
   seed     = ReverseBitOrder(seed, polynomialOrder);        // reverse bits order of the seed value
   CRCWDATL = (unsigned short)(seed&0x0000FFFF);             // set seed value
   CRCWDATH = (unsigned short)(seed>>16);


   _CRCIF = 0;                                               // clear interrupt flag
   switch(polynomialOrder)                                   // load dummy data to shift out the
                                                             // seed result
   {
        case 8:
           *((unsigned char*)&CRCDATL) = 0;                  // load byte
           while(!_CRCIF);                                   // wait until shifts are done
           seed = CRCWDATL&0x00ff;                           // read reversed seed
        case 16:
           CRCDATL = 0;                                      // load short
           while(!_CRCIF);                                   // wait until shifts are done
           seed = CRCWDATL;                                  // read reversed seed
           break;
        case 32:
           // load long
           CRCDATL = 0;
           CRCDATH = 0;
           while(!_CRCIF);                                   // wait for shifts are done
           seed = ((unsigned long)CRCWDATH<<16)|CRCWDATL;    // read reversed seed
           break;
        default:
           ;
   }

   seed = ReverseBitOrder(seed, polynomialOrder);            // reverse the bit order to get the
                                                             // non-direct seed
   return seed;                                              // return the non-direct CRC initial value
}
```

Example 5-2: Calculating 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;
}
```

5.4.4 CRC RESULT

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

In Alternate mode (MOD bit = 1, not available on all devices), the result is available in the CRCWDAT registers 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 bit = 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 registers. After the shifts are finished, the final CRC result can be read from the CRCWDAT registers.

There are two procedures to get the final CRC result after all data is loaded into the CRC module.

If the data width (DWIDTH<4:0>) is more than the polynomial length (PLEN<4:0>):

1. Wait for the data FIFO to empty (CRCMPT bit is set).
2. Wait (DWIDTH<4:0> + 1) clock cycles to make sure that shifts from the shift buffer are finished.
3. Change the data width to the polynomial length (DWIDTH<4:0> = PLEN<4:0>).
4. Write one dummy data word to the CRCDAT registers.
5. Wait two peripheral clock cycles to move the data from the FIFO to the shift buffer, plus (PLEN<4:0>+ 1) clock cycles to shift out the result.

   Alternatively, clear the CRC Interrupt Selection bit (CRCISEL = 0) to get the interrupt when all shifts are done. Clear the CRC interrupt flag. Write dummy data in the CRCDAT registers and wait until the CRC interrupt flag is set.
6. Read the final CRC result from the CRCWDAT registers.
7. Restore the data width (DWIDTH<4:0> bits) for further calculations (OPTIONAL).

If the data width (DWIDTH<4:0>) is equal to, or less than, the polynomial length (PLEN<4:0>), the procedure to get the result can be different:

1. Clear the CRC Interrupt Selection bit (CRCISEL = 0) to get the interrupt when all shifts are done.
2. Suspend the calculation by setting CRCGO = 0.
3. Clear the CRC interrupt flag.
4. Write the dummy data with the total data length equal to the polynomial length in the CRCDAT registers.
5. Resume the calculation by setting CRCGO = 1.
6. Wait until the CRC interrupt flag is set.
7. Read the final CRC result from the CRCWDAT registers.

When the CRC result is achieved, the CRC non-direct initial value should be written again into the CRCWDAT registers to clear/reset the shift buffer from the previously loaded dummy data to start a new calculation. Example 5-3 shows the steps described above for the polynomial orders of 8, 16 and 32 bits.

Example 5-3: Routine to Get the Final CRC Result in Legacy Mode (MOD bit = 0)

```
unsigned long  GetCRC(unsigned char polynomialOrder,          // valid values are 8,16,32
unsigned char  currentDataWidth)                              // valid values are 8,16,32
{
unsigned long  crc = 0;

    while(!CRCCON1bits.CRCMPT);                               // wait until data FIFO is empty

    asm   volatile ("repeat %0\n nop" : : "r"(currentDataWidth>>1)); // wait until previous data
                                                              // shifts are done
    CRCCON2bits.DWIDTH  = polynomialOrder-1;                  // set data width to polynomial
                                                              // length
    CRCCON1bits.CRCISEL = 0;                                  // interrupt when all shifts are done


    _CRCIF = 0;                                               // clear interrupt flag

    switch(polynomialOrder)
    {
        case 8:                                               // polynomial length is 8 bits
           *((unsigned char*)&CRCDATL) = 0;                   // load byte
           while(!_CRCIF);                                    // wait until shifts are done
           crc = CRCWDATL&0x00ff;                             // get crc
           break;
        case 16:                                              // polynomial length is 16 bits
           CRCDATL = 0;                                       // load short
           while(!_CRCIF);                                    // wait until shifts are done
           crc = CRCWDATL;                                    // get crc
           break;
        case 32:                                              // polynomial length is 32 bits
           CRCDATL = 0;                                       // load long
           CRCDATH = 0;
           while(!_CRCIF);                                    // wait until shifts are done
           crc = ((unsigned long)CRCWDATH<<16)|CRCWDATL;      // get crc
           break;
        default:
           ;
    }
    CRCCON2bits.DWIDTH = currentDataWidth-1;                  // restore data width for further
                                                              // calculations


    return crc;                                               // return the final CRC value

}
```

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.

For more details on interrupts and interrupt priority settings, refer to the **"Interrupt Controller"** section in the device data sheet.

6.0 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.

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.

6.2 Typical Operation

To use the module for a typical CRC calculation:

1. Set the CRCEN bit to enable the module.
2. Configure the module for the desired operation:
   a) Program the desired polynomial using the CRCXOR registers 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 registers as described in **Section 5.4.3 “CRC Initial Value”**.
5. Load all data into the FIFO by writing to the CRCDAT registers 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 5.4.4 “CRC Result”**.

Example 6-1 through Example 6-10 show typical code for different combinations of polynomial length, data width, shift direction and CRC Engine modes.

Example 6-1: 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 6-2: CRC-SMBus (8-Bit Polynomial with 32-Bit Data, Little-Endian, MOD bit = 0)

```
// ASCII bytes "12345678"
volatile unsigned  char __attribute__((aligned(2))) message[] = {'1','2','3','4','5','6','7','8'};

volatile unsigned  char crcResultCRCSMBUS = 0;

int      main     (void)
{
unsigned short*   pointer;
unsigned short    length;
unsigned short    data_high;
unsigned short    data_low;
////////////////////////////////////////////////////////////////////////////
// standard CRC-SMBUS
////////////////////////////////////////////////////////////////////////////

#define  CRCSMBUS_POLYNOMIAL  ((unsigned short)0x0007)
#define  CRCSMBUS_SEED_VALUE  ((unsigned short)0x0000)// non-direct of 0x00

    CRCCON1 = 0;
    CRCCON2 = 0;

    CRCCON1bits.CRCEN  = 1;                          // enable CRC
    CRCCON1bits.LENDIAN = 0;                         // big endian
    CRCCON1bits.CRCISEL = 0;                         // interrupt when all shifts are done
    CRCCON2bits.DWIDTH  = 32-1;                      // 32-bit data width
    CRCCON2bits.PLEN   = 8-1;                        // 8-bit polynomial order
    CRCCON1bits.CRCGO  = 1;                          // start CRC calculation

    CRCXORL = CRCSMBUS_POLYNOMIAL;                   // set polynomial
    CRCXORH = 0;


    CRCWDATL = CRCSMBUS_SEED_VALUE;                  // set initial value
    CRCWDATH = 0;


    pointer = (unsigned short*)message;              // calculate CRC
    length  = sizeof(message)/sizeof(unsigned long);
    while(length--)
    {
    while(CRCCON1bits.CRCFUL);                       // wait if FIFO is full

        data_low  = *pointer++;                      // load from little endian
        data_high = *pointer++;

        asm     volatile ("swap %0" : "+r"(data_low));  // swap bytes for big endian
        asm     volatile ("swap %0" : "+r"(data_high));

        CRCDATL = data_high;                         // 32-bit word access to FIFO
        CRCDATH = data_low;                          // swap 16-bit words for big endian
    }

    while(!CRCCON1bits.CRCMPT);                      // wait until FIFO is empty

                                                     // wait until previous data shifts are done
    asm  volatile ("repeat #16-#2\n nop");           // 16 cycles maximum for 32-bit data width

    CRCCON2bits.DWIDTH = 8-1;                        // 8-bit
                                                     // switch data width to polynomial length

    _CRCIF = 0;                                      // clear the interrupt flag
                                                     // dummy data to shift out the CRC result

    *((unsigned char*)&CRCDATL) = 0;                 // byte access to FIFO

    while(!_CRCIF);                                  // wait until shifts are done
    crcResultCRCSMBUS = CRCWDATL&0x00ff;             // get CRC result (must be 0xC7)

    while(1);

    return 1;
}
```

Example 6-3: 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 6-4: 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 short)0x8005)
#define  CRC16_SEED_VALUE ((unsigned short)0x0000) // non-direct of 0x0000

    CRCCON1 = 0;
    CRCCON2 = 0;

    CRCCON1bits.CRCEN   = 1;                    // enable CRC
    CRCCON1bits.CRCISEL = 0;                    // interrupt when all shifts are done
    CRCCON1bits.LENDIAN = 1;                    // little endian
    CRCCON2bits.DWIDTH  = 16-1;                 // 16-bit data width
    CRCCON2bits.PLEN    = 16-1;                 // 16-bit polynomial order
    CRCCON1bits.CRCGO   = 1;                    // start CRC calculation

    CRCXORL = CRC16_POLYNOMIAL;                 // set polynomial
    CRCXORH = 0;

    CRCWDATL = CRC16_SEED_VALUE;                // set initial value
    CRCWDATH = 0;

    pointer = (unsigned short*)message;         // calculate CRC
    length  = sizeof(message)/sizeof(unsigned short);

    while(length--)
    {

        while(CRCCON1bits.CRCFUL);              // wait if FIFO is full


        data = *pointer++;                      // load data


        CRCDATL = data;                         // 16-bit word access to FIFO
    }

        while(CRCCON1bits.CRCFUL);              // wait if FIFO is full
    CRCCON1bits.CRCGO = 0;                      // suspend CRC calculation to clear interrupt flag

    _CRCIF  = 0;                                // clear interrupt flag

    CRCDATL = 0;                                // load dummy data to shift out the CRC result
                                                // data width must be equal to polynomial length


    CRCCON1bits.CRCGO = 1;                      // resume CRC calculation

    while(!_CRCIF);                             // wait until shifts are done


    crcResultCRC16 = CRCWDATL;                  // get CRC result (must be 0xE716)

    while(1);

    return 1;
}
```

Example 6-5: 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 6-6: CRC-CCITT (16-Bit Polynomial with 16-Bit Data, Big-Endian, MOD bit = 0)

```
// 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 short)0x1021)
#define  CRCCCITT_SEED_VALUE ((unsigned short)0x84CF) // non-direct of 0xffff

    CRCCON1 = 0;
    CRCCON2 = 0;

    CRCCON1bits.CRCEN   = 1;                    // enable CRC
    CRCCON1bits.CRCISEL = 0;                    // interrupt when all shifts are done
    CRCCON1bits.LENDIAN = 0;                    // big endian
    CRCCON2bits.DWIDTH  = 16-1;                 // 16-bit data width
    CRCCON2bits.PLEN    = 16-1;                 // 16-bit polynomial order
    CRCCON1bits.CRCGO   = 1;                    // start CRC calculation

    CRCXORL = CRCCCITT_POLYNOMIAL;              // set polynomial
    CRCXORH = 0;

    CRCWDATL = CRCCCITT_SEED_VALUE;             // set initial value
    CRCWDATH = 0;

    pointer = (unsigned short*)message;         // calculate CRC
    length  = sizeof(message)/sizeof(unsigned short);

    while(length--)
    {

        while(CRCCON1bits.CRCFUL);              // wait if FIFO is full


        data = *pointer++;                      // load data


        asm  volatile ("swap %0" : "+r"(data));  // swap bytes for big endian
    CRCDATL = data;                             // 16 bit word access to FIFO
    }

    while(CRCCON1bits.CRCFUL);                  // wait if FIFO is full

    CRCCON1bits.CRCGO = 0;                      // suspend CRC calculation to clear interrupt flag


    _CRCIF = 0;                                 // clear interrupt flag


    CRCDATL = 0;                                // load dummy data to shift out the CRC result
                                                // data width must be equal to polynomial length

    CRCCON1bits.CRCGO = 1;                      // resume CRC calculation


    while(!_CRCIF);                             // wait until shifts are done


    crcResultCRCCCITT = CRCWDATL;               // get CRC result (must be 0x9B4D)

    while(1);

    return 1;
}
```

Example 6-7: 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 6-8: 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   long crcResultCRC32 = 0;

int main(void)
{
unsigned  short*   pointer;
unsigned  short    length;
////////////////////////////////////////////////////////////////////////////
// standard CRC-32
////////////////////////////////////////////////////////////////////////////
#define   CRC32_POLYNOMIAL    ((unsigned long)0x04C11DB7)
#define   CRC32_SEED_VALUE    ((unsigned long)0x46AF6449)   // non-direct of 0xffffffff

    CRCCON1 = 0;
    CRCCON2 = 0;

    CRCCON1bits.CRCEN   = 1;                                // enable CRC
    CRCCON1bits.CRCISEL = 0;                                // interrupt when all shifts are done
    CRCCON1bits.LENDIAN = 1;                                // little endian
    CRCCON2bits.DWIDTH  = 32-1;                             // 32-bit data width
    CRCCON2bits.PLEN    = 32-1;                             // 32-bit polynomial order
    CRCCON1bits.CRCGO   = 1;                                // start CRC calculation

    CRCXORL = CRC32_POLYNOMIAL&0x0000ffff;                  // set polynomial
    CRCXORH = CRC32_POLYNOMIAL>>16;

    CRCWDATL = CRC32_SEED_VALUE&0x0000ffff;                 // set initial value
    CRCWDATH = CRC32_SEED_VALUE>>16;

    pointer = (unsigned short*)message;                     // calculate CRC
    length  = sizeof(message)/sizeof(unsigned long);
    while(length--)
    {

        while(CRCCON1bits.CRCFUL);                          // wait if FIFO is full

                                                            // 32-bit word access to FIFO
        CRCDATL = *pointer++;                               // must be written first
        CRCDATH = *pointer++;                               // must be written last
    }

    while(CRCCON1bits.CRCFUL);                              // wait if FIFO is full
```

Example 6-8: CRC-32 (32-Bit Polynomial with 32-Bit Data, Little-Endian, MOD bit = 0) (Continued)

```
   CRCCON1bits.CRCGO = 0;                                  // suspend CRC calculation to clear interrupt flag
   _CRCIF = 0;                                             // clear interrupt flag


   CRCDATL = 0;                                            // dummy data to shift out the CRC result
   CRCDATH = 0;

   CRCCON1bits.CRCGO = 1;                                  // resume CRC calculation


   while(!_CRCIF);                                         // wait until shifts are done


   crcResultCRC32 = ((unsigned long)CRCWDATH<<16)|CRCWDATL; // get the final CRC result


   crcResultCRC32 = ~ReverseBitOrder(crcResultCRC32);      // OPTIONAL
                                                           // reverse CRC value bit order and
                                                           // invert (must be 0x9AE0DAAF)

   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 6-9: 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 6-10: 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 short*    pointer16;
unsigned short     length;

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

    CRCCON1 = 0;
    CRCCON2 = 0;

    CRCCON1bits.CRCEN   = 1;                            // enable CRC
    CRCCON1bits.LENDIAN = 1;                            // little endian
    CRCCON2bits.DWIDTH  = 32-1;                         // 32-bit data width
    CRCCON2bits.PLEN    = 32-1;                         // 32-bit polynomial order
    CRCCON1bits.CRCGO   = 1;                            // start CRC calculation

    CRCXORL = CRC32_POLYNOMIAL&0x0000ffff;              // set polynomial
    CRCXORH = CRC32_POLYNOMIAL>>16;

    CRCWDATL = CRC32_SEED_VALUE&0x0000ffff;             // set initial value
    CRCWDATH = CRC32_SEED_VALUE>>16;

    pointer16 = (unsigned short*)message1;              // calculate CRC
    length    = sizeof(message1)/sizeof(unsigned long);
    while(length--)
    {

        while(CRCCON1bits.CRCFUL);                      // wait if FIFO is full
                                                        // 32-bit word access to FIFO
        CRCDATL = *pointer16++;                         // must be written first
        CRCDATH = *pointer16++;                         // must be written last
    }

                                                        // wait until previous
                                                        // data shifts are done
    while(!CRCCON1bits.CRCMPT);                         // wait until FIFO is empty

    asm     volatile    ("repeat #16-#2\n nop");        // 16 cycles maximum for 32-bit data

    CRCCON2bits.DWIDTH = 8-1;                           // switch the data width to 8-bit

    pointer8 = (unsigned char*)message2;                // calculate CRC
    length   = sizeof(message2)/sizeof(unsigned char);
    while(length--)
    {
```

Example 6-10: Data Width Switching (32-Bit Polynomial, Little-Endian, MOD bit = 0) (Continued)

```
        while(CRCCON1bits.CRCFUL);                    // wait if FIFO is full

        *((unsigned char*)&CRCDATL) = *pointer8++;    // byte access to FIFO
   }

   while(!CRCCON1bits.CRCMPT);                        // wait until FIFO is empty

                                                      // wait until previous data shifts are done
asm     volatile    ("repeat #4-#2\n nop");           // 4 cycles maximum for 8-bit data

                                                      // switch the data width to polynomial length
   CRCCON2bits.DWIDTH = 32-1;                         // 32-bit


   CRCDATL = 0;                                       // dummy data to shift out the CRC result
   CRCDATH = 0;

   asm  volatile    ("repeat #2+#16-#2\n nop");       // delay 2 cycles to move data from FIFO
                                                      // to shift buffer
                                                      // and 16 cycles for 32-bit word to shift out
                                                      // the final result

   crcResultCRC32 = ((unsigned long)CRCWDATH<<16)|CRCWDATL; // get the final CRC result
                                                      // (must be0xE092727E)

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

7.0 OPERATION IN POWER SAVE MODES

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.

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.

8.0 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 dsPIC33/PIC24 device families, 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 dsPIC33/PIC24 families of devices.

9.0 REVISION HISTORY

Revision A (April 2009)

This is the initial released revision of this document.

Revision B (August 2013)

This revision includes the following changes:

- Changed the document name from PIC24F Family Reference Manual to dsPIC33/PIC24 Family Reference Manual.
- Revised description of CRCISEL in Register 3-1.
- Added additional information to **Section 5.2 “Data Shift Direction”**.
- Added additional information to **Section 5.3 “Data FIFO”**.
- Made corrections to Figure 5-1, Figure 5-2 and Figure 5-3.
- Revised **Section 5.4 “CRC Engine Interface”**.
- Revised **Section 5.5 “Interrupt Operation”** and added code examples.
- Revised **Section 6.2 “Typical Operation”** and added code examples.
- Minor grammatical corrections throughout the document.

Revision C (May 2018)

This revision includes the following changes:

- Revised Table 3-1.
- Added note to Register 3-1.
- Added note to Figure 4-1.
- Removed Figure 5-1, Figure 5-2 and Figure 5-3.
- Revised **Section 5.4.3 “CRC Initial Value”**, **Section 5.4.4 “CRC Result”** and **Section 5.5 “Interrupt Operation”**.
- Revised title for Example 5-3,

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: 978-1-5224-2981-4

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

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
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

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

China - Beijing
Tel: 86-10-8569-7000

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

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

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

China - Guangzhou
Tel: 86-20-8755-8029

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

China - Hong Kong SAR
Tel: 852-2943-5100

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

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

China - Shanghai
Tel: 86-21-3326-8000

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

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

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

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

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

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

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

India - Pune
Tel: 91-20-4121-0141

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

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

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

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

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

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

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

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

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

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

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

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

Finland - Espoo
Tel: 358-9-4520-820

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

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-67-3636

Germany - Karlsruhe
Tel: 49-721-625370

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

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

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

Italy - Padova
Tel: 39-049-7625286

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

Norway - Trondheim
Tel: 47-7289-7561

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

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

Sweden - Gothenberg
Tel: 46-31-704-60-40

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

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