**TIMER 0**

In the real-time clock lab we are required to use TIMER0 to generate a 10ms interrupt. Let us look at what is required. First let us review the features of TIMER0.

(PIC18F27/47/57Q43 datasheet DS40002147D)

**Introduction**

**24. TMR0 - Timer0 Module**

The Timer0 module has the following features:

• 8-bit timer with programmable period

• 16-bit timer

• Selectable clock sources

• Synchronous and asynchronous operation

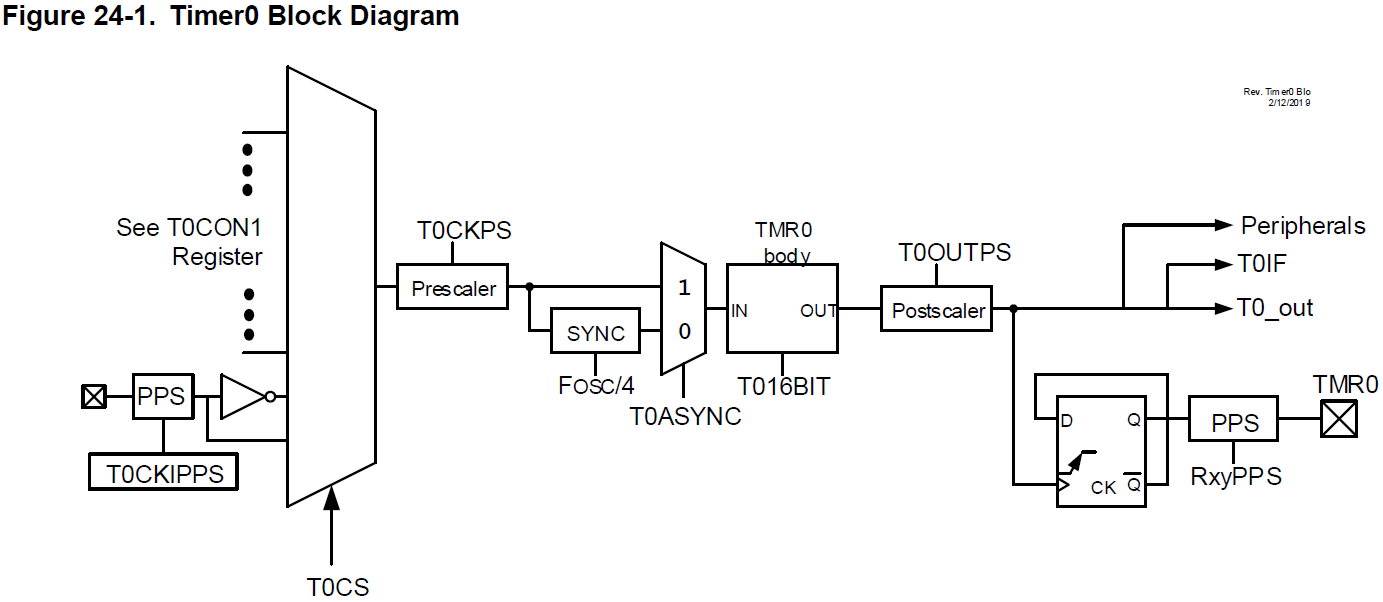
• Programmable prescaler (Independent of Watchdog Timer)

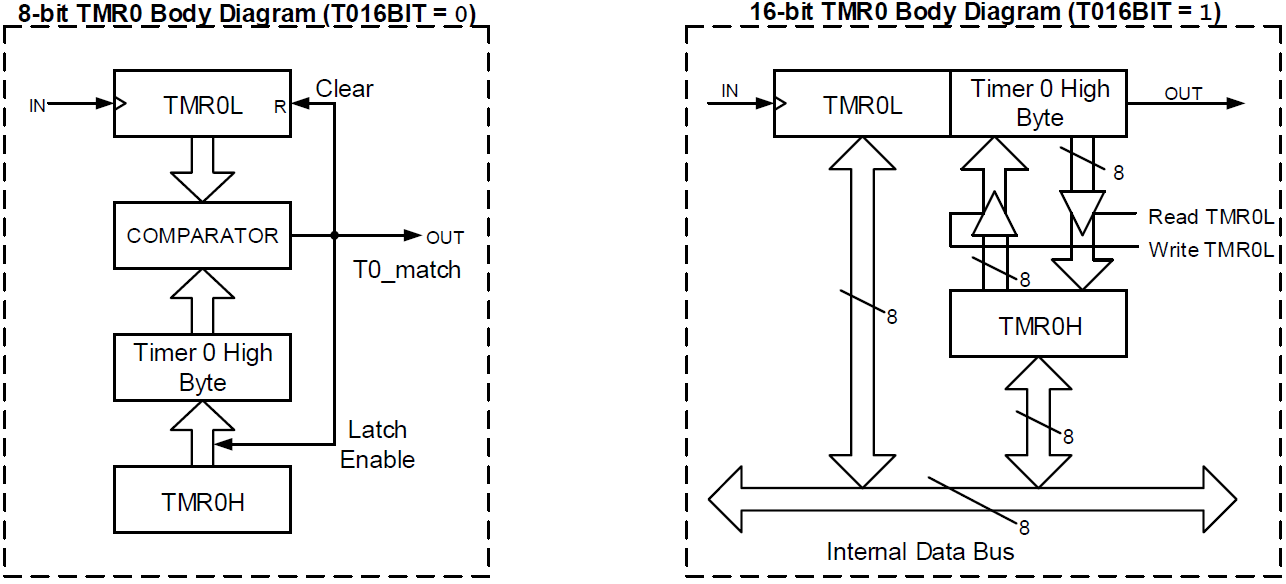
• Programmable postscaler

• Interrupt on match or overflow

• Output on I/O pin (via PPS) or to other peripherals

• Operation during Sleep





**Enable timer0 (T0CON0 Bit 7 – EN TMR0 Enable):**

**24.1 Timer0 Operation**

Timer0 can operate as either an 8-bit or 16-bit timer. The mode is selected with the MD16 bit.

**Set timer0 to 8-bit mode (T0CON0 Bit 4 – MD16 16-Bit Timer Operation Select):**

**24.1.1 8-Bit Mode**

In this mode Timer0 increments on the rising edge of the selected clock source. A prescaler on the clock input gives several prescale options (see prescaler control bits, CKPS). In this mode, as shown in Figure 24-1, a buffered version of TMR0H is maintained.

This is compared with the value of TMR0L on each cycle of the selected clock source. When the two values match, the following events occur:

• TMR0L is reset

• The contents of TMR0H are copied to the TMR0H buffer for next comparison

**Set clock selection to MFINTOSC (500 kHz) (T0CON1 Bits 7:5 – CS[2:0] Timer0 Clock Source):**

**24.2 Clock Selection**

Timer0 has several options for clock source selections, the option to operate synchronously/asynchronously and an available programmable prescaler. The CS bits are used to select the clock source for Timer0.

**Set Clock to synchronous mode (T0CON1 Bit 4 – ASYNC TMR0 Input Asynchronization Enable):**

**24.2.1 Synchronous Mode**

When the ASYNC bit is clear, Timer0 clock is synchronized to the system clock (FOSC/4). When operating in Synchronous mode, Timer0 clock frequency cannot exceed FOSC/4. During Sleep mode the system clock is not available and Timer0 cannot operate.

**Set prescaler to 1:2 (T0CON1 Bits 3:0 – CKPS[3:0] Prescaler Rate Select):**

**24.2.3 Programmable Prescaler**

Timer0 has 16 programmable input prescaler options ranging from 1:1 to 1:32768. The prescaler values are selected using the CKPS bits. The prescaler counter is not directly readable or writable. The prescaler counter is cleared on the following events:

• A write to the TMR0L register

• A write to either the T0CON0 or T0CON1 registers

• Any device Reset

**Set Postscaler to 1:10 (T0CON0 Bits 3:0 – OUTPS[3:0]):**

**24.2.4 Programmable Postscaler**

Timer0 has 16 programmable output postscaler options ranging from 1:1 to 1:16. The postscaler values are selected using the OUTPS bits. The postscaler divides the output of Timer0 by the selected ratio. The postscaler counter is not directly readable or writable. The postscaler counter is cleared on the following events:

• A write to the TMR0L register

• A write to either the T0CON0 or T0CON1 registers

• Any device Reset

**Load TMR0H with 249 (TMR0H):**

**24.3 Timer0 Output and Interrupt**

**24.3.1 Timer0 Output**

TMR0\_out toggles on every match between TMR0L and TMR0H in 8-bit mode, or when TMR0H:TMR0L rolls over in 16-bit mode. If the output postscaler is used, the output is scaled by the ratio selected. The Timer0 output can be routed to an I/O pin via the RxyPPS output selection register, or internally to a number of Core Independent Peripherals. The Timer0 output can be monitored through software via the OUT output bit.

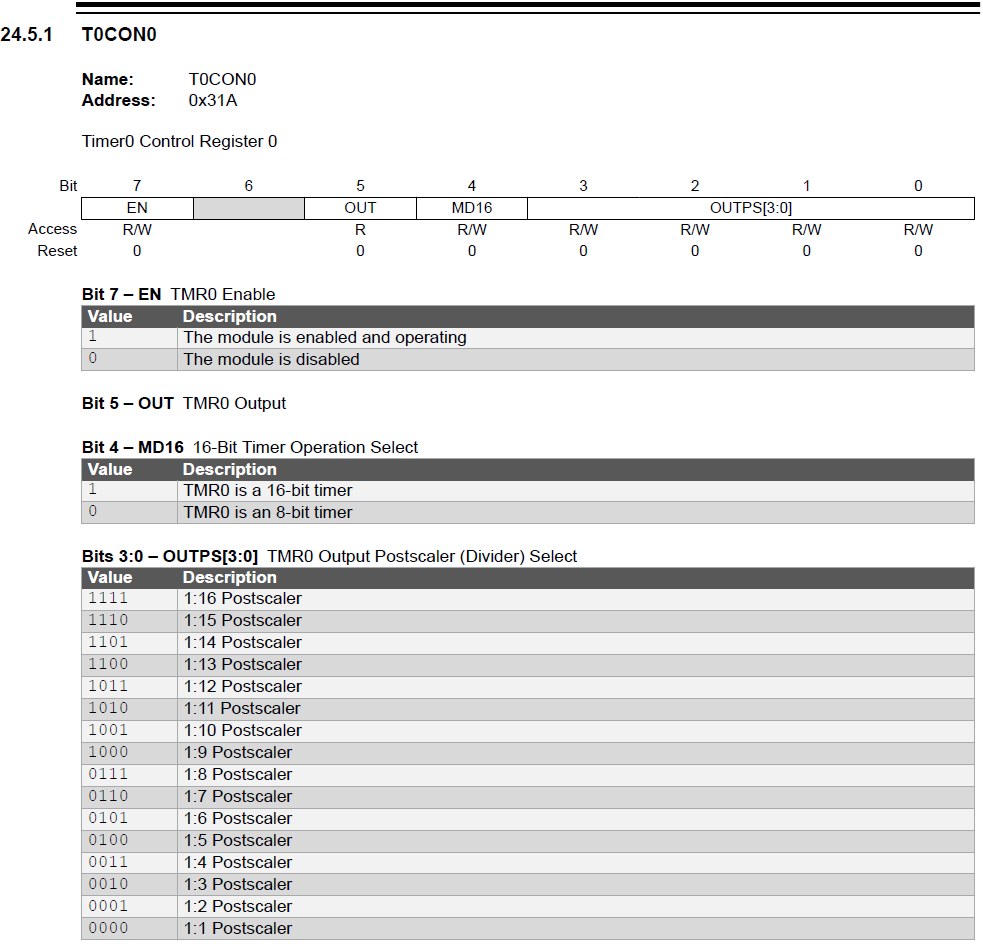
**Setup timer0 interrupt (TMR0IE):**

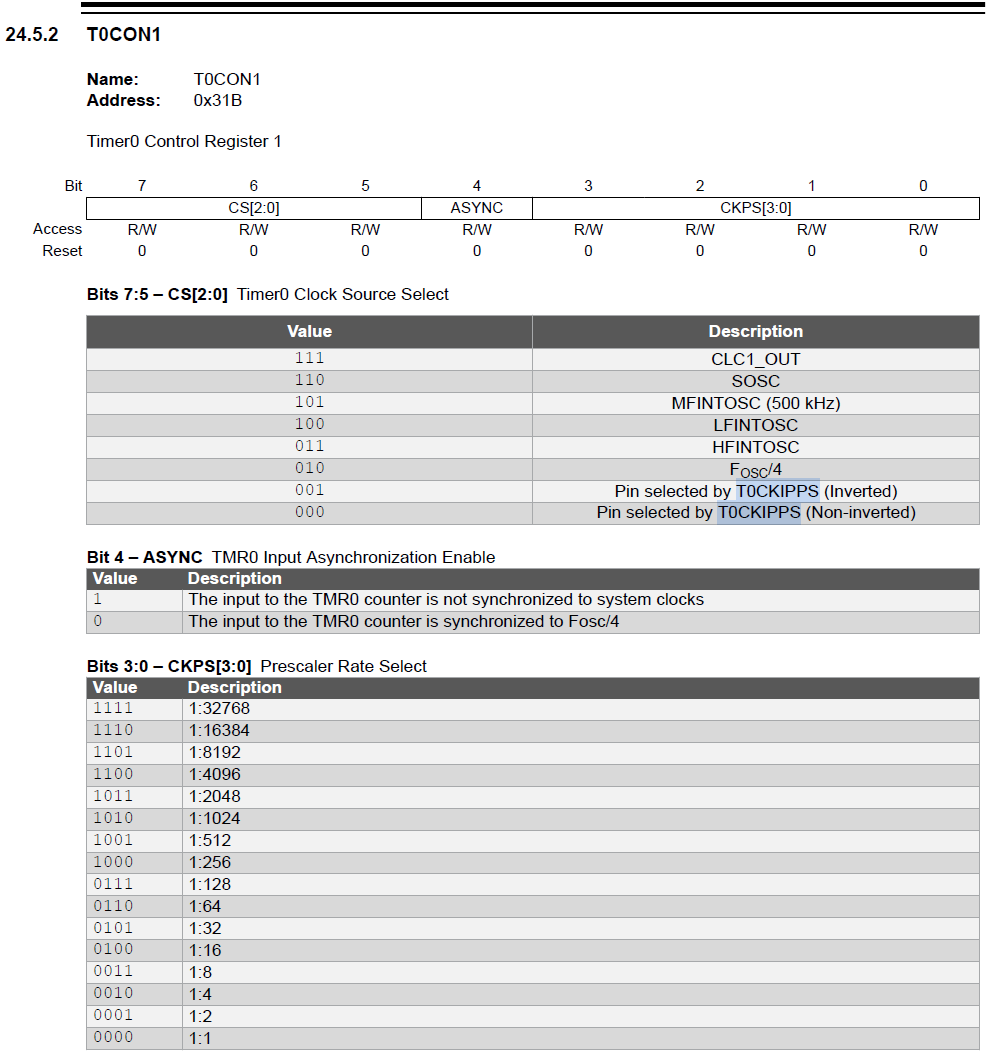
**24.3.2 Timer0 Interrupt**

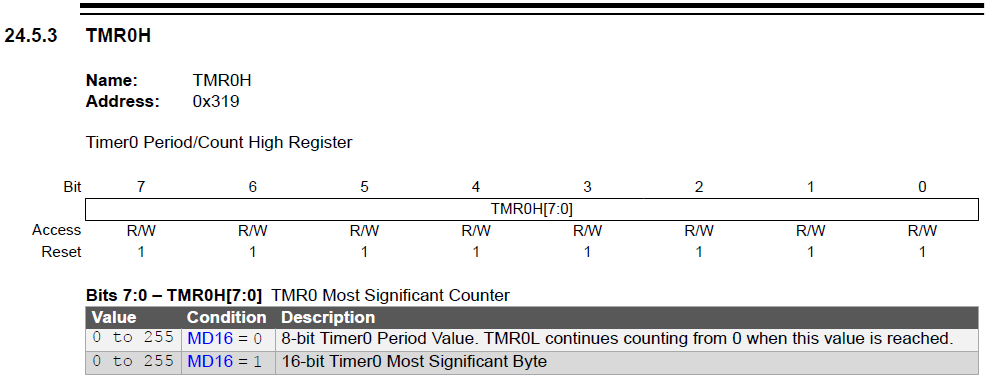
The Timer0 Interrupt Flag bit (TMR0IF) is set when the TMR0\_out toggles. If the Timer0 interrupt is enabled (TMR0IE), the CPU will be interrupted when the TMR0IF bit is set. When the postscaler bits (T0OUTPS) are set to 1:1 operation (no division), the T0IF flag bit will be set with every TMR0 match or rollover. In general, the TMR0IF flag bit will be set every T0OUTPS +1 matches or rollovers.

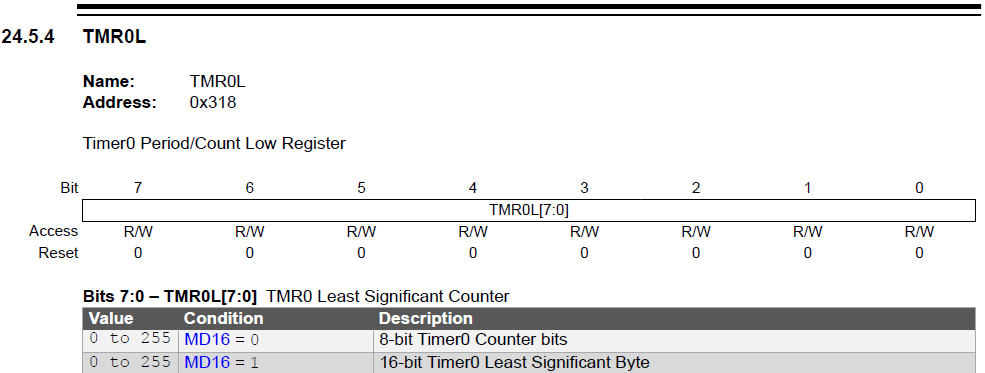
Notice all the capabilities of this hardware. It is well suited to our application. If you want to design an accurate real-time clock you would attach a crystal clock or temperature compensated clock circuit to T0CKIPPS input. For this lab we will use the built in 500 kilohertz RC Clock circuit.

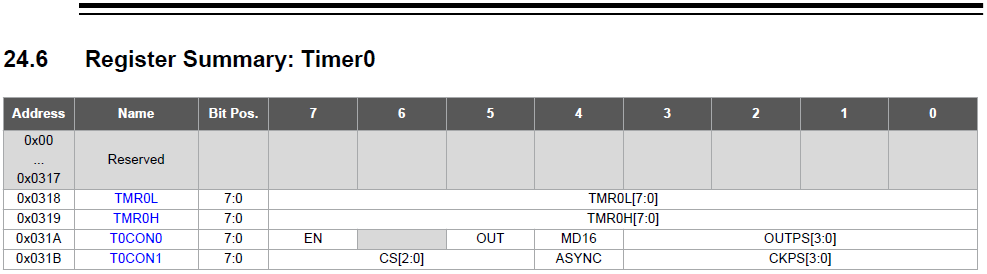
We need to setup the registers of this timer to generate a 10ms timer interrupt.











**11. VIC - Vectored Interrupt Controller Module**

**11.1 Overview**

The Vectored Interrupt Controller (VIC) module reduces the numerous peripheral interrupt request signals to a single interrupt request signal to the CPU. This module includes the following major features:

• Interrupt Vector Table (IVT) with a unique vector for each interrupt source

• Fixed and ensured interrupt latency

• Programmable base address for IVT with lock

• Two user-selectable priority levels - High priority and low priority

• Two levels of context saving

• Interrupt state Status bits to indicate the current execution status of the CPU

The VIC module assembles all of the interrupt request signals and resolves the interrupts based on both a fixed natural order priority (i.e., determined by the IVT), and a user-assigned priority (i.e., determined by the IPRx registers), thereby eliminating scanning of interrupt sources.

**11.2 Interrupt Control and Status Registers**

The devices in this family implement the following registers for the interrupt controller:

• INTCON0, INTCON1 Control Registers

• PIRx - Peripheral Interrupt Status Registers

• PIEx - Peripheral Interrupt Enable Registers

• IPRx - Peripheral Interrupt Priority Registers

Global interrupt control functions and external interrupts are controlled from the **INTCON0** register. The **INTCON1** register contains the status flags for the interrupt controller.

The **PIRx** registers contain all of the interrupt request flags. Each source of interrupt has a Status bit, which is set by the respective peripherals or an external signal, and is either cleared via software or automatically cleared by hardware upon clearing of the interrupt condition, depending on the peripheral and bit.

The **PIEx** registers contain all of the interrupt enable bits. These control bits are used to individually enable interrupts from the peripherals or external signals.

The **IPRx** registers are used to set the interrupt priority level for each source of interrupt. Each user interrupt source can be assigned to either a high or low priority.

**11.5 Interrupt Operation**

All pending interrupts are indicated by their respective flag bit being equal to a ‘1’ in the PIRx register. All pending interrupts are resolved using the priority scheme explained in Interrupt Priority section.

Once the interrupt source to be serviced is resolved, the program execution vectors to the resolved interrupt vector addresses, as explained in Interrupt Vector Table section. The vector number is also stored in the WREG register. Most of the flag bits are required to be cleared by the application software, but in some cases, device hardware clears the interrupt automatically. Some flag bits are read-only in the PIRx registers. These flags are a summary of the source interrupts and the corresponding interrupt flags of the source must be cleared.

A valid interrupt can be either a high- or low-priority interrupt when in the main routine or a high-priority interrupt when in a low priority Interrupt Service Routine. Depending on the order of interrupt requests received and their relative timing, the CPU will be in a state of execution indicated by STAT bit

The state machine shown in Figure 11-1 and the subsequent sections detail the execution of interrupts when received in different orders.

**Important:** The state of GIEH/L is not changed by the hardware when servicing an interrupt. The internal

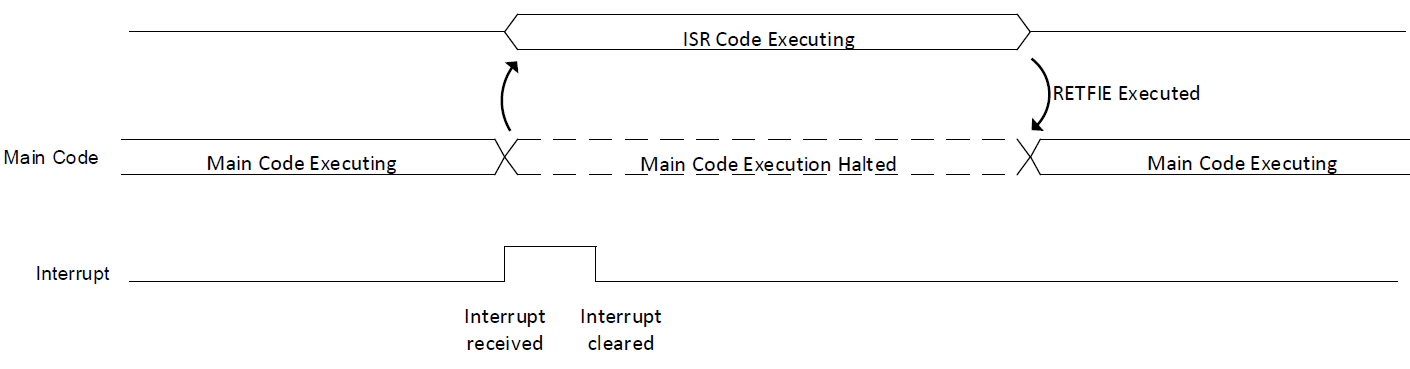
state machine is used to keep track of execution states. These bits can be manipulated in the user code,

resulting in transferring execution to the main routine and ignoring existing interrupts.

**11.5.1 Serving a High- or Low-Priority Interrupt while the Main Routine Code is Executing**

When a high- or low-priority interrupt is requested while the main routine code is executing, the main routine execution is halted and the ISR is addressed. Upon a return from the ISR (by executing the **RETFIE** instruction), the main routine resumes execution.

**Figure 11-2. Interrupt Execution: High/Low-Priority Interrupt while Executing Main Routine**



**11.6 Context Saving**

The interrupt controller supports a two-level deep context saving system (main routine context and low ISR context). Refer to the state machine shown in Figure 11-6 for details.

The Program Counter (PC) is saved on the dedicated device PC stack. The CPU registers saved include STATUS, WREG, BSR, FSR0/1/2, PRODL/H and PCLATH/U.

After WREG has been saved to the context registers, the resolved vector number of the interrupt source to be serviced is copied into WREG. Context save and restore operation is completed by the interrupt controller based on the current state of the interrupts and the order in which they were sent to the CPU.

Context save/restore works the same way in both states of MVECEN. When IPEN = 0, there is only one level of interrupt active. Hence, only the main context is saved when an interrupt is received.

**11.7 Returning from Interrupt Service Routine (ISR)**

The “Return from Interrupt” instruction (RETFIE) is used to mark the end of an ISR.

When the RETFIE 1 instruction is executed, the PC is loaded with the saved PC value from the top of the PC stack. Saved context is also restored with the execution of this instruction. Thus, execution returns to the state of operation that existed before the interrupt occurred.

When the RETFIE 0 instruction is executed, the saved context is not restored back to the registers.

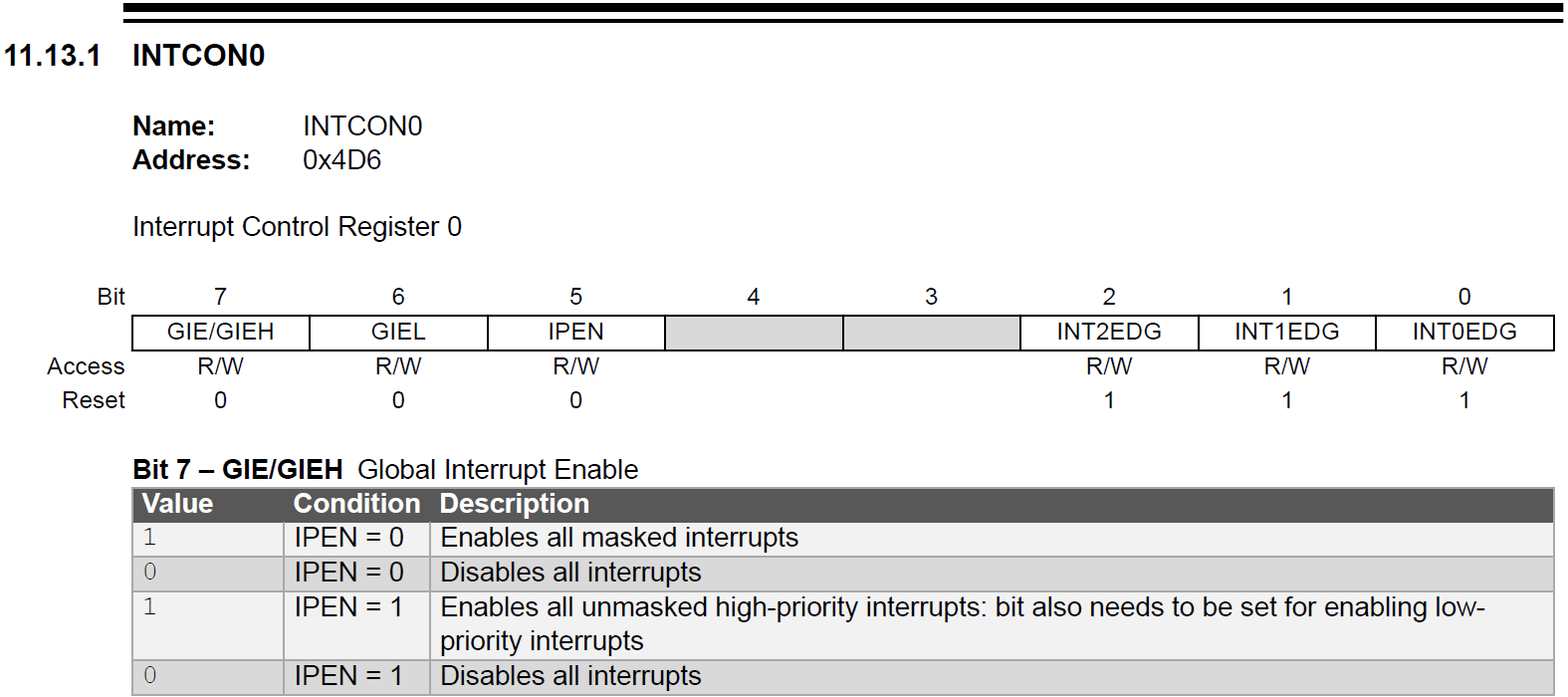
**11.8 Interrupt Latency**

When MVECEN = 1, there is a fixed latency of three instruction cycles between the completion of the instruction active when the interrupt occurred, and the first instruction of the Interrupt Service Routine. Figure 11-7, Figure 11-8, and Figure 11-9 illustrate the sequence of events when a peripheral interrupt is asserted, when the last executed instruction is one-cycle, two-cycle and three-cycle, respectively.

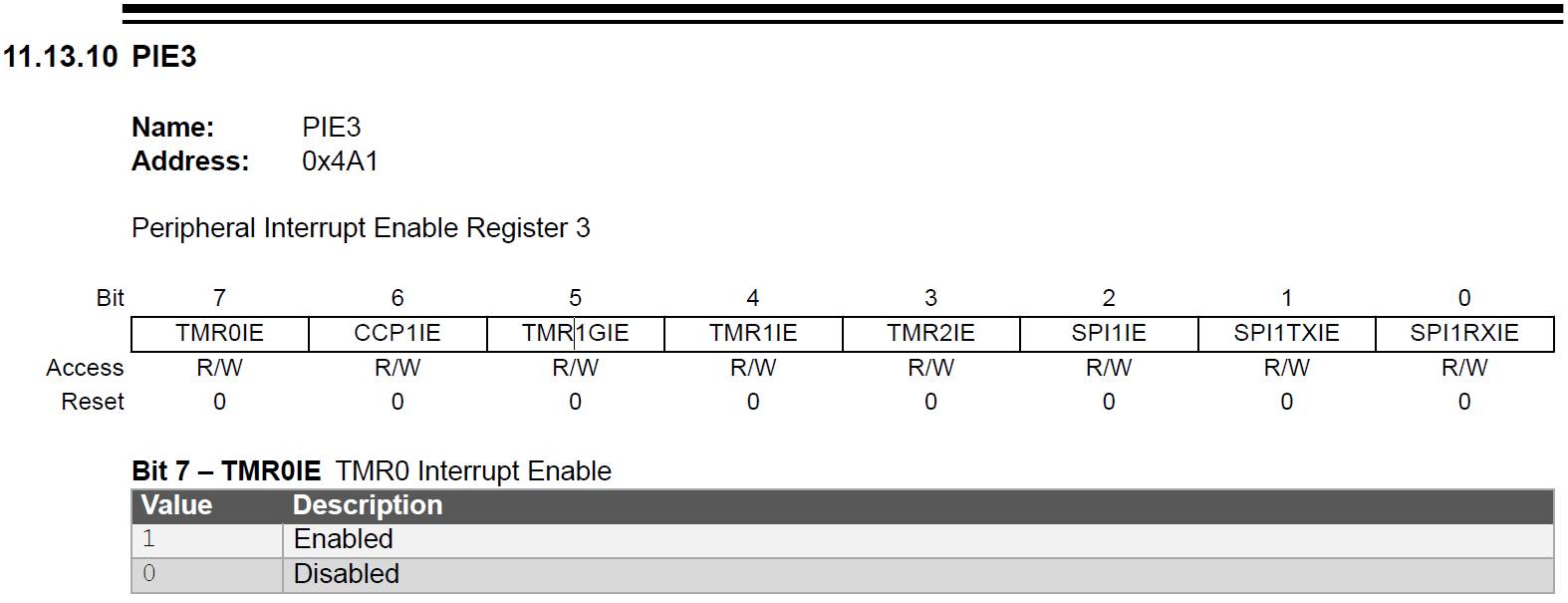
After the Interrupt Flag Status bit is set, the current instruction completes executing. In the first latency cycle, the contents of the PC, STATUS, WREG, BSR, FSR0/1/2, PRODL/H and PCLATH/U registers are context saved and the IVTBASE + Vector number is calculated. In the second latency cycle, the PC is loaded with the calculated vector table address for the interrupt source and the starting address of the ISR is fetched. In the third latency cycle, the PC is loaded with the ISR address. All the latency cycles are executed as NOP instructions.

When MVECEN = 0, the interrupt controller requires two clock cycles to vector to the ISR from the main routine. Note that as this mode requires additional software to determine which interrupt source caused the interrupt, the actual latency between the trigger and the beginning of the specific ISR for each individual interrupt will be longer than two clock cycles and will vary, when not using vectored interrupts.

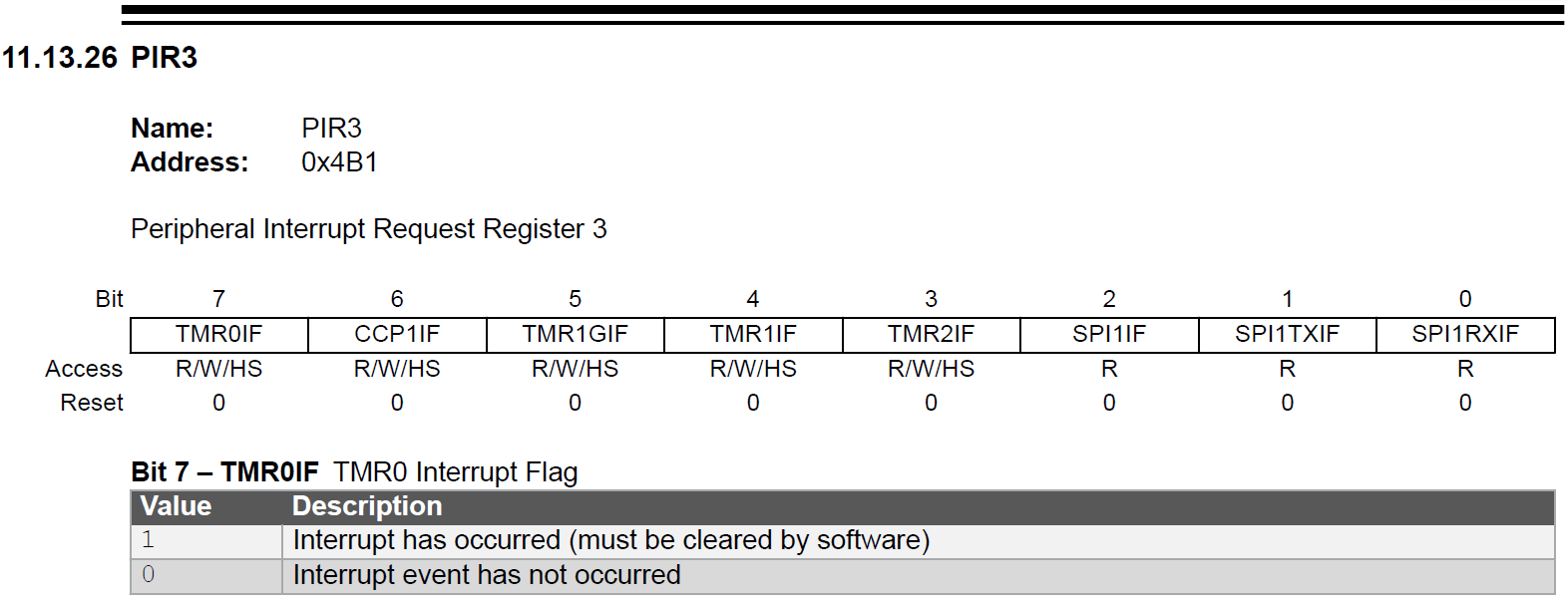
**Set Global Interrupt Enable (GIEH):**



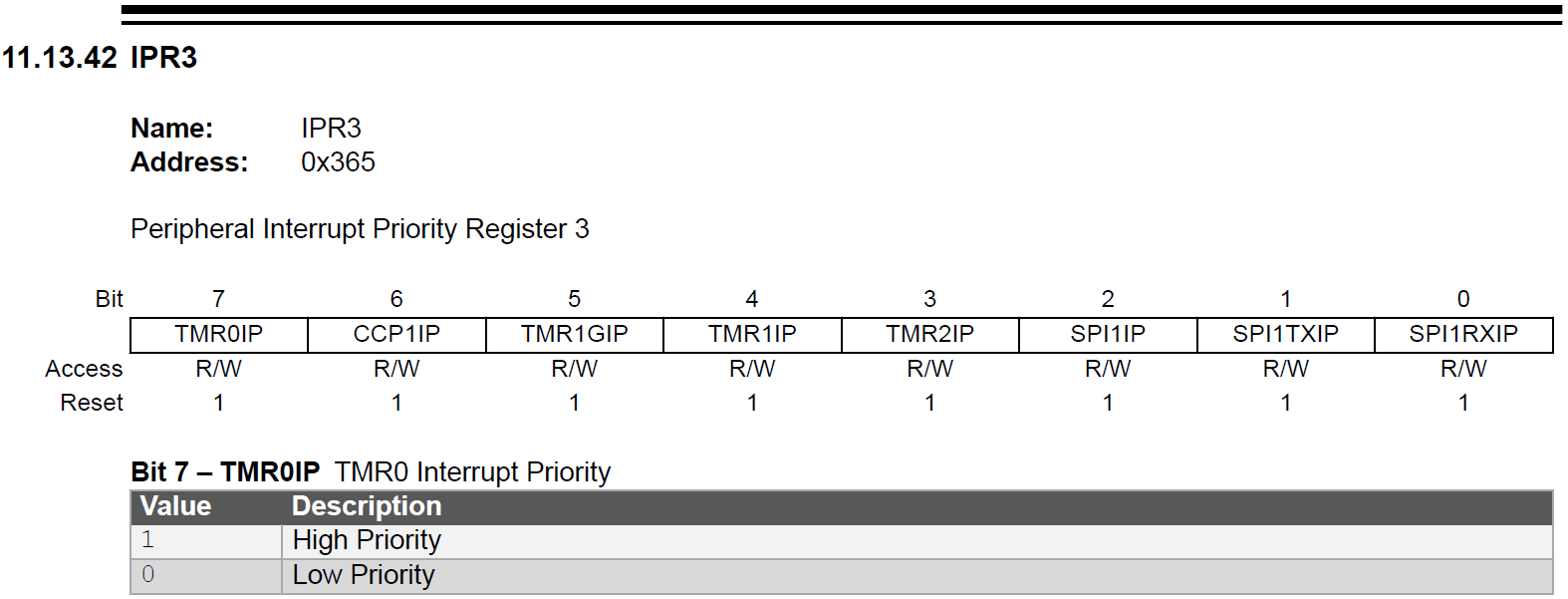
**Set timer0 Interrupt Enabled (TMR0IE):**



**Clear timer0 Interrupt Flag before enabling timer0 interrupt (TMR0IF):**



**Set timer0 Peripheral Interrupt Priority to High Priority (TMR0IP):**



**11.14 Register Summary – Interrupts:**

****

****

****

****

****

**12.1.2.2 MFINTOSC**

The Medium-Frequency Internal Oscillator (MFINTOSC) generates two constant clock outputs (500 kHz and 31.25 kHz). The MFINTOSC clock signals are created from the HFINTOSC using dynamic divider logic, which provides constant MFINTOSC clock rates regardless of selected HFINTOSC frequency.

The MFINTOSC cannot be used as the system clock, but can be used as a clock source for certain peripherals, such as a Timer.

We need to calculate a 10ms interrupt from the information in the datasheet shown above:

Clock source (500 kHz) / Prescaler (1:2) / Postscaler (1:10) / Period (250)

500 kHz / 2 / 10 / 250 = 100 Hz

1/100 Hz = 10ms

Write a function called timerInit() to initialize timer0 to produce a 10ms interrupt. Use the following coments and the information in the datasheet shown above to complete the correct code:

void timerInit(void)

{

// tmr0 enable, 8 bit, 1:10 Postscaler

// 500 kHz, sync, 1:2 prescaler

// TMR0H 0 - 249 count for a period of 250

// TMR0 high priority interrupt

// clear TMR0 interrupt flag

// TMR0 interrupt enable

// enable global interrupt

}

But wait a minute, in order to respond to the interrupt, we will need to execute some code. So we now need to look at the code that will need to be in the ISR.

(taken from (MPLAB® XC8 C Compiler User’s Guide for PIC® MCU 50002737C))

**5.8 Interrupts**

The MPLAB XC8 compiler incorporates features allowing interrupts to be fully handled from C code. Interrupt functions are often called Interrupt Service Routines, or ISRs.

The operation of interrupts is handled differently by the different device families. Most Baseline devices do not implement interrupts at all; Mid-range devices have one vector location which is linked to all interrupt sources; some PIC18 devices have two independent interrupt vectors, one assigned to low-priority interrupt sources, the other to high-priority sources; and some PIC18 devices implement a vectored interrupt controller (VIC) module with support for one or more interrupt vector tables (IVTs), which can be populated with the addresses of high- or low-priority interrupt functions.

The operation of the IVT on devices with a VIC module can be disabled by clearing the MVECEN configuration bit. The device is then said to be operating in legacy mode, operating with dual priorities and dual vector locations. This bit is also used by the compiler to determine how interrupt functions should be programmed. Although the vector table is disabled in this mode, the vector locations are still relocatable. By default the vector location will be 0x8 and 0x18, the same for regular PIC18 devices without the VIC module.

The priority scheme implemented by PIC18 devices can also be disabled by clearing the IPEN SFR bit. Such devices are then said to be operating in Mid-range compatibility mode and utilize only one interrupt vector, located at address 0x8.

Interrupt functions must not be called directly from C code (due to the different return instruction that is used), but interrupt functions can call other functions, both user-defined and library functions.

Interrupt code is the name given to any code that executes as a result of an interrupt occurring, including functions called from the ISR and library code. Interrupt code completes at the point where the corresponding return from interrupt instruction is executed. This contrasts with main-line code, which, for a freestanding application, is usually the main part of the program that executes after Reset.

**5.8.1 Writing an Interrupt Service Routine**

The prototype and content of an ISR will vary based on the target device and the project being compiled. Observe the following guidelines when writing an ISR.

For devices that do not have the VIC module:

• Write each ISR prototype using the \_\_interrupt() specifier.

• Use void as the return type and for the parameter specification.

• If your device supports interrupt priorities, with each function use the low\_priority (or \_\_low\_priority) or high\_priority (or \_\_high\_priority) arguments to \_\_interrupt().

• Inside the ISR body, determine the source of the interrupt by checking the interrupt flag and the interrupt enable for each source that is to be processed and make the relevant interrupt code conditional on those being set.

For devices which are using the VIC module:

• Write each ISR prototype using only the \_\_interrupt() specifier.

• Use void as the return type and specify a parameter list of either void or one char argument if you need to identify the interrupt source.

• As arguments to the \_\_interrupt() specifier in the ISR prototype, specify which sources each interrupt function should handle, using either irq() or \_\_irq(); specify the interrupt priority assigned to the function’s source, using either low\_priority (or \_\_low\_priority) or high\_priority (or \_\_high\_priority); and optionally, specify the base address of the IVT in which to place the function’s address, using either base() (or \_\_base()).

• If the ISR processes more than one source, determine the source of the interrupt from the function’s parameter, if specified, or by checking the interrupt flag and the interrupt enable for each source that is to be processed.

For all devices:

• Inside the ISR body, clear the relevant interrupt flag once the source has been processed.

• Do not re-enable interrupts inside the ISR body. This is performed automatically when the ISR returns.

• Keep the ISR as short and as simple as possible. Complex code will typically use more registers that will increase the size of the context switch code.

**An example of an interrupt function:**

Here is the function code, split and modified for a device using vector tables. Since only one interrupt source is associated with each ISR, the interrupt code does not need to determine the source and is therefore faster.

void \_\_interrupt(irq(TMR0),high\_priority) tc0Int(void)

{

TMR0IF=0;

++tick\_count;

return;

}

void \_\_interrupt(irq(TMR1),high\_priority) tc1Int(void)

{

TMR1IF=0;

tick\_count += 100;

return;

}

Use the following code for our project:

void \_\_interrupt(irq(TMR0),high\_priority) tickINT(void)

{

TMR0IF=0;

tickCount++;

return;

}

void \_\_interrupt(irq(default),high\_priority) defaultINT(void)

{

return;

}