CHAPTER 6

# Working with time: interrupts, counters and timers

Apart from the lucky few, our daily lives are ruled by time. We have alarm clocks to wake us in the morning, stopwatches to measure time duration, timers to start off single events (such as a VCR recording) and timers to maintain periodic events (such as a house heating system coming on at the same time every day). For the young and those who teach, the working day is ruled by a school timetable – a complex series of timed events.

For embedded systems, time is similarly of the essence. At a simple level the system needs to respond in a timely manner to external events. It may also need to measure time between external events and generate time-based activity. These requirements are met primarily by two differing, but related, features of a microcontroller: the interrupt and the counter/timer. While each is a stand-alone element in its own right, both are so useful that they have become ubiquitous, finding their way into many other microcontroller features. Interrupts are to be found generated by almost every microcontroller peripheral. Counters/timers provide the timing for a range of activity, from motor control with pulse width modulation to baud rate generation in serial communications. The principles of each are introduced in this chapter and need to be understood with care. Because they are used both at a simple level and in advanced and sophisticated ways, we return to them on a number of occasions throughout the book. Ultimately, we find that interrupts and counters/timers all form part of the exciting techniques that underpin real-time programming.

In this chapter you will learn about:

- Why we need interrupts and counters/timers.
- The underlying interrupt hardware structure.
- The 16F84A interrupt structure.
- How to program with interrupts.
- The underlying microcontroller counter/timer hardware structure.
- The 16F84A Timer 0 structure.
- Simple applications of the counter/timer.
- The Sleep mode.

Designing Embedded Systems with PIC Microcontrollers; ISBN: 9781856177504

If you wish you will also be able to get deeper into 16F84A interrupt issues, in particular its interrupt latency.

## 6.1 The main idea – interrupts

As we know, a computer CPU is a deeply orderly entity, following the instructions of the program one by one and doing what it is told in a precise and predictable fashion. An interrupt disturbs this order. Coming maybe when least expected, its function is to alert the CPU in no uncertain terms that some significant external event has happened, to stop it from what it is doing and force it (at the greatest speed possible) to respond to what has happened. Originally interrupts were applied to allow emergency external events, such as power failure, the system overheating or major failure of a subsystem to get the attention of the CPU. But the concept of interrupts was recognised as being very powerful. As time went on, more and more subsystems gained the power to generate interrupts. This forced increasing complexity in interrupt structures and a need to recognise that not all interrupts were equal.

To work successfully with interrupts, we need to understand both the hardware interrupt structure and the programming techniques needed to program successfully with them. An introduction to these now follows.

### *6.1.1 Interrupt structures*

Different microcontrollers have rather different interrupt structures. Inevitably they have more than one interrupt source, usually with some internally generated and others external. A generic structure, which illustrates the main hardware principles, is shown in Figure 6.1. On the left we see one of several sources, ‘Interrupt X’. If an interrupt occurs, it sets an S–R bistable. The occurrence of the interrupt, even if it is only momentary, is thus recorded. The output of the bistable, the latched version of the interrupt, is called the ‘interrupt flag’. This is then gated with an enable signal, ‘Interrupt X Enable’. If this is high, then the interrupt signal progresses to an OR gate. If it is low, the interrupt signal gets no further. If enabled, it is

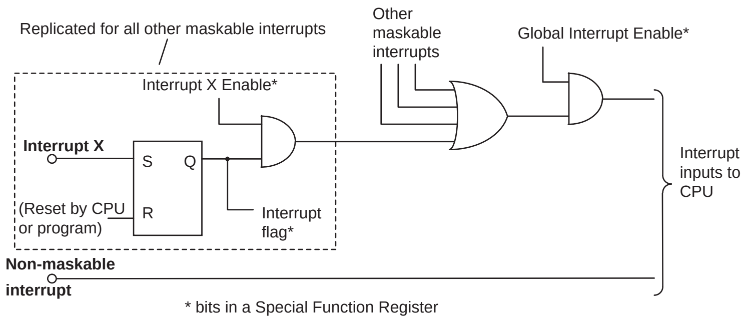

Replicated for all other maskable interrupts
Other maskable interrupts
Global Interrupt Enable*
Interrupt X Enable*
Interrupt X
S
Q
R
(Reset by CPU or program)
Interrupt flag*
Interrupt inputs to CPU
Non-maskable interrupt
* bits in a Special Function Register

Figure 6.1: A simple generic interrupt structure

ORed with other enabled interrupt inputs of the microcontroller. The OR gate output will go high if *any* interrupt input is high. There is then a further gating of the OR gate output, this time with a 'Global Interrupt Enable'. Only if that value is high can any interrupt signal reach the CPU. When the CPU has responded to an interrupt, it is necessary to clear the interrupt flag. In some processors this is done automatically by the CPU, in others it must be done within the program.

The action of disabling an interrupt is sometimes called 'masking'. It seems strange, however, to be able to switch off a capability which is so important and which is meant to be there to report emergencies. Therefore, some microcontrollers have interrupts that cannot be masked. These are always external (i.e. not from an internal peripheral) and are used to connect to external interrupt signals of the greatest importance. A non-maskable interrupt is shown in Figure 6.1. As the CPU always responds if it occurs, there is less point in storing it as a flag, and this is sometimes therefore not done.

### 6.1.2 The 16F84A interrupt structure

The 16F84A has four interrupt sources, all of which can be individually enabled or disabled:

- *External interrupt*. This is the only external interrupt input. It shares a pin with Port B, bit 0 (Figure 2.1). It is edge triggered.
- *Timer overflow*. This is an interrupt caused by the Timer 0 module, which is the subject of the second half of this chapter. It occurs when the timer's 8-bit counter overflows.
- *Port B interrupt on change*. This interrupts when a change is detected on any of the higher four bits of Port B. The mechanism was described in Section 3.4.1.
- *EEPROM write complete*. This interrupts when a write instruction to the EEPROM memory is completed.

The interrupt structure is shown in Figure 6.2 and the SFR that controls it, **INTCON**, in Figure 6.3. It is useful to study the two diagrams in parallel, as every bit in the **INTCON** register appears in the structure logic diagram. The four sources appear labelled on the left of Figure 6.2. When comparing this diagram with Figure 6.1, it is interesting to note the absence of the interrupt flag flip-flops. These exist, but are not shown in the Microchip diagram. Each source has an enable line (labelled …**E**) and a flag line (labelled …**F**).

Thus, the lines **TOIF**, **INTF** and so on are actually the interrupt flags rather than the interrupt inputs themselves. All can be seen as bits in the **INTCON** register, with the exception of the EEPROM write complete flag and enable. Note that the external interrupt is edge triggered. The edge it responds to is controlled by the setting of the **INTEDG** bit of the **OPTION** register (shown later, as it mainly relates to Timer 0, in Figure 6.9).

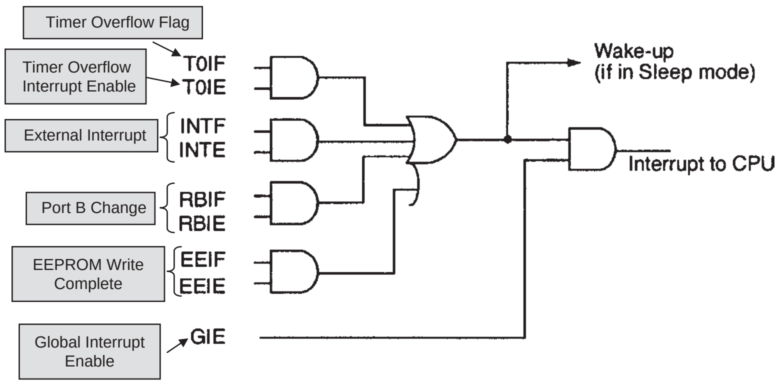


**Figure 6.2: The 16F84A interrupt structure (supplementary labels in shaded boxes added by the author)**

As in Figure 6.1, each flag is ANDed with a corresponding Enable input (**TOIE**, **INTE**, **RBIE** and **EEIE**). The enable bits are located in the **INTCON** register and can be set by the programmer. The outputs of the four AND gates are then ORed together, before passing on to the Global Enable gate. Interrupt flags must be cleared by manipulating their **INTCON** bits in the program. The 16F84A has no non-maskable interrupt input.

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-x |
|---|---|---|---|---|---|---|---|
| GIE | EEIE | TOIE | INTE | RBIE | TOIF | INTF | RBIF |
| bit 7 | | | | | | | bit 0 |

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

**Figure 6.3: The 16F84A *INTCON* register**

## *6.1.3 The CPU response to an interrupt*

Let us assume that an interrupt has occurred, and both its local enable and the global enable are set. The interrupt is therefore detected by the CPU and it executes a special section of program called the Interrupt Service Routine (ISR). It is important to understand the underlying detail of what goes on, and this is illustrated in the flow diagram of Figure 6.4. The CPU completes the instruction it is currently executing and saves the value of the Program Counter on the top of the Stack. Thus, it will 'know' where to come back to when the ISR is complete. To avoid other interrupts possibly interrupting this interrupt, it also clears the Global Interrupt Enable.

In the PIC 16 Series, the ISR *must* start at the interrupt vector, program memory location 0004 (Figure 2.4). Therefore, when an interrupt occurs, this value is loaded into the Program Counter and program execution then continues from the reset vector. In any processor, the ISR *must* end with a special 'return from interrupt' instruction. In the 16 Series this is the **retfie** instruction. When this is detected, the CPU sets the **GIE** to 1, loads the Program Counter from the top of the Stack and then resumes program execution. Thus, it returns to the instruction which follows the instruction during which the interrupt was detected.

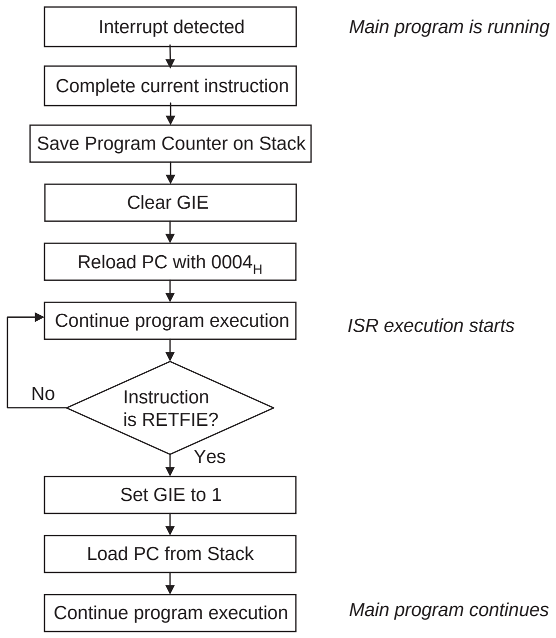


**Figure 6.4: The 16F84A interrupt response sequence of events**

## 6.2 Working with interrupts

### *6.2.1 Programming with a single interrupt*

It is comparatively easy to write simple programs with just one interrupt. For success, the essential points to watch are:

- Start the ISR at the interrupt vector, location 0004.
- Enable the interrupt that is to be used by setting the enable bit in the **INTCON** register.
- Set the Global Enable bit, **GIE**.

```
;*******************************************************
;Int_Demo1
;This program demonstrates simple interrupts.
;Intended for simulation.
;tjw rev.14.2.09            Tested in simulation 14.9.09
;*******************************************************
;
       include p16f84A.inc
;Port A all output
;Port B: bit 0 = Interrupt Input
;
       org    00
       goto   start
;
       org    04     ;here if interrupt occurs
       goto   Int_Routine
;
       org    0010
;Initialise
start  bsf    status,rp0   ;select bank 1
       movlw  01
       movwf  trisb        ;portb bits 1-7 output
                           ;      bit 0 is input
       movlw  00
       movwf  trisa        ;porta bits all output
;Comment in or out following instruction to change
;interrupt edge
;      bcf    option_reg,intedg
       bcf    status,rp0  ;select bank 0
       bsf    intcon,inte ;enable external interrupt
       bsf    intcon,gie  ;enable global int

wait   movlw  0a     ;set up initial port output values
       movwf  porta
       nop
       movlw  15
       movwf  porta
       goto   wait
;
       org    0080
Int_Routine
       movlw  00
       movwf  porta
       bcf    intcon,intf  ;clear the interrupt flag
       retfie
       end
```

**Program Example 6.1: Simple interrupt application**

- Clear the interrupt flag within the ISR.
- End the ISR with a **retfie** instruction.
- Ensure that the interrupt source, for example Port B or Timer 0, is actually set up to generate interrupts!

Program Example 6.1 gives a very simple interrupt example, intended for simulation. The interrupt guidelines in the list above are applied. The program starts as usual at the reset vector 0000; however, the interrupt vector is now also in use. The first action of the program is to branch over the reset vector to location **start**, where initialisation takes place. Within this we see the **GIE** and **INTE** bits being set. These bit labels can be used because they appear in the 16F84A Include File. The main program simply outputs the bit patterns $0A_H$ and $15_H$ to Port A in turn.

When an interrupt occurs the interrupt vector address is loaded into the Program Counter, from where program execution continues. The first action of the ISR is to jump to location **Int_Routine**. This is placed at program memory location $0080_H$ to give clarity to the simulation. The ISR simply clears Port A before clearing its interrupt flag and returning to the main program.

**Programming Exercise 6.1**

Copy Program Example 6.1 from the book's companion website into MPLAB and create a project around it. Build the project and enable the simulator. Open a Watch window, displaying **PORTA, PORTB, INTCON** and **PCL**. Open the Hardware Stack window (under **View**) to observe the contents of the Stack. Open a new stimulus workbook (as described in Section 4.8.1) and set Pin **RB0** to Toggle. Single-step through the program, and observe and understand the change to each observed variable on every instruction. Now 'fire' the **RB0** pin, setting **RB0** high, and an interrupt sequence should be instigated as you single-step further. See the Hardware Stack change, program execution transfer to the ISR and on ISR completion the program resuming after the instruction where it was interrupted. Fire the **RB0** pin again (returning it to 0), and continue stepping. This will cause no change to program execution, as the interrupt edge response will be positive-edge triggered only (the **INTEDG** bit has been left at a Reset value of 1). Try changing the **INTEDG** bit (to 0), as shown in the program, to change the edge to which the interrupt responds.

When you have implemented the above program successfully, try inserting the errors below into the program. They are commonly made by novices. Observe and explain the effect.

- Fail to clear the interrupt flag by removing the instruction **bcf intcon,intf**.
- Terminate the ISR incorrectly by replacing **retfie** with **return**.

### 6.2.2 Moving to multiple interrupts – identifying the source

Using one interrupt in a program is generally quite easy, but be warned – once we start using more than one, interrupts can interact with each other in ways which are far from simple. Complexity then seems to rise approximately in proportion to the *square* of the number of interrupts used!

As we have seen, the 16F84A has four interrupt sources but only one interrupt vector. Therefore, if more than one interrupt is enabled, it is not obvious at the beginning of an ISR which interrupt has occurred. In this case the programmer must write the ISR so that at its beginning it tests the flags of all possible interrupts and determines from this which one has been called. An example piece of code that does this, assuming all four interrupt sources are enabled, is shown in Program Example 6.2.

```
interrupt btfsc intcon,0      ;test RBIF
       goto portb_int
       btfsc intcon,1         ;test external interrupt flag
       goto ext_int
       btfsc intcon,2         ;test timer overflow flag
       goto timer_int
       btfsc eecon1,4         ;test EEPROM write complete flag
       goto eeprom_int

portb_int
...
place portb change ISR here
...
       bcf    intcon,0        ;and clear the interrupt flag
       retfie

ext_int
...
place external interrupt ISR here
...
       bcf    intcon,1        ;and clear the interrupt flag
       retfie

timer_int
...
place timer overflow ISR goes here
...
       bcf    intcon,2        ;and clear the interrupt flag
       retfie
       eeprom_int
...
place EEPROM write complete ISR here
...
       bcf    eecon1,4        ;and clear the interrupt flag
       retfie
```

**Program Example 6.2: Interrupt source identification**

### 6.2.3 Stopping interrupts from wrecking your program 1 – context saving

Because an interrupt can occur at any time, it has the power to be extremely destructive. Program Example 6.3 is written to illustrate this. It applies a 16-bit addition subroutine.

For the purposes of the example, the 16-bit number $9999_H$ is added to itself, with an expected 17-bit result $13332_H$. In the subroutine, the lower two bytes, **qlo** and **plo**, are added. Any Carry generated is then added into one of the higher bytes, and the higher two bytes are added. An ISR is written which affects both the Carry flag and the W register, as most ISRs would.

```
;*****************************************************************
;Int_context
;This program demonstrates the need for context saving.
;Intended for simulation.

;TJW 15.4.05                                  Tested 17.4.05
;****************************************************************
;Port A not used. Port B bit 0 used for externalinterrupt input
       #include p16f84A.inc
;
rhi          equ   10
rlo          equ   11
phi          equ   12
plo          equ   13
qhi          equ   14
qlo          equ   15

             org 00
             goto start
             org 04 ;here if interrupt occurs
             goto Int_Routine ;
start        org 0010
             bsf    intcon,inte ;enable external interrupt
             bsf    intcon,gie  ;enable global int
loop         movlw 99
             movwf phi    ;preload numbers to be added
             movwf plo
             movwf qhi
             movwf qlo
             call Double_add
             movlw 00     ;clear result
             movwf rhi
             movwf rlo
             goto  loop
;This subroutine adds two 16-bit numbers, stored in phi-plo, and qhi-qlo,
;and stores result in rhi-rlo. 16-bit overflow in Carry flag at end.
Double_add
             movf  plo,0         ;move plo to the W reg
             addwf qlo,0         ;add lower bytes
             movwf rlo
             btfsc status,0


             incf  phi,1         ;add in Carry
             movf  phi,0
             addwf qhi,0         ;add upper bytes
             movwf rhi
             return
Int_Routine
             bcf   status,0      ;clear the Carry flag
             movlw 0ff           ;change W reg value
             bcf   intcon,intf
             retfie
             end
```

**Program Example 6.3: Impact of interrupts**

Suppose the interrupt occurs immediately after the first subroutine **movf** instruction, where the W register is holding the value of **plo**. The ISR changes the W register so, when program execution returns to the subroutine, it will be with the incorrect W register value. Suppose the interrupt occurs immediately after the first **addwf** instruction. The value of the Carry bit is essential to the success of the addition, but again is lost in the ISR.

**Programming Exercise 6.2**

Copy the program Int_Context from the book's companion website into MPLAB and create a project around it. Build the project and enable the simulator. Open a Watch window, displaying **qhi**, **qlo**, **phi**, **plo**, **rhi**, **rlo**, **STATUS** and **WREG**. Open the Stimulus Controller and set Pin **RB0** to Toggle. Single-step through the program, and check that the addition works correctly and the expected result is achieved. Now try inserting interrupts at different points in the program. Note how at many points the occurrence of the ISR destroys the validity of the addition's result.

The temporary data being used in a particular activity in the CPU is called its 'context'. In the PIC 16 Series this includes at least the W register value and the Status register. It is clearly important to save the context when an interrupt occurs. Some microcontrollers do this automatically, but PIC 16 Series microcontrollers do not. Therefore, it is up to the programmer to ensure that whatever context saving that is needed is done in the program.

Program Example 6.4 shows the recommended Microchip method for saving the W register into a pre-designated memory location **W_TEMP** and the Status register into a location called **STATUS_TEMP**. The **swapf** and **movwf** instructions are used because they do not affect any Status register bits.

```
PUSH   movwf w_temp         ;Copy W to W_TEMP register,
       swapf status,0       ;Swap status to be saved into W
       movwf status_temp    ;Save status to STATUS_TEMP register
ISR                         ;Interrupt Service Routine
...
       actual ISR goes here
...
POP    swapf status_temp,0  ;Swap nibbles in STATUS_TEMP register
                                        ;and place result into W
       movwf status         ;Move W into STATUS register ;sets bank to original
                                        ;state


       swapf w_temp,1       ;Swap nibbles in W_TEMP and keep result in W_TEMP
       swapf w_temp,0       ;Swap nibbles in W_TEMP and place result into W
...
       clear interrupt flag(s) here
...
       retfie
```

**Program Example 6.4: Context saving**

**Programming Exercise 6.3**

Adapt the context saving shown in Program Example 6.4 and insert it into the Int_Context program (Program Example 6.3). You will need to define memory locations for **w_temp** and **status_temp**. Check that the program now operates correctly wherever you force an interrupt to occur.

### *6.2.4 Stopping interrupts from wrecking your program 2 – critical regions and masking*

We can resolve *some* of the problems of an interrupt occurring in a program section like the subroutine discussed above by appropriate context saving. Unfortunately, we can't resolve them all, at least not just with context saving.

What if an interrupt occurred in a software delay routine, for example that of Program Example 5.2? The delay length would be increased by the duration of the ISR, which could be disastrous, and no amount of context saving would improve the situation.

Consider a more subtle problem. The ISR shown in Program Example 6.5 takes the word held in **rhi-rlo**, calculated in the subroutine of Program Example 6.3, and outputs it to a 12-bit digital-to-analog converter (DAC) connected to Ports A and B. We assume that the overall program constrains the word in **rhi-rlo** to 12 bits. Suppose the ISR shown in Program Example 6.5 occurs during the subroutine of Example 6.3. Context saving is implemented, so should there be a problem?

Unfortunately, there is a problem. The ISR is making use of a result that is being calculated in a program section that it is interrupting. Suppose **rlo** has just been updated and not **rhi** when the interrupt occurs. The ISR outputs the new value of **rlo** and the old one of **rhi**. Together, they might make a number that has no sense, with potentially disastrous consequences.

```
Int_Routine
       movwf W_temp      ;Copy W to TEMP register,
       swapf status,0    ;Swap status to be saved into W
       movwf status_temp ;Save status to STATUS_TEMP register
       bcf   status,5    ;ensure we are in Bank 0
       movf  rhi,0       ;output higher 4 bits to DAC
       movwf porta
       movf  rlo,0          ;output lower 8 bits to DAC
       movwf portb
       swapf status_temp,0  ;Swap nibbles in STATUS_TEMP register
                                         ;and place result into W
       movwf status         ;Move W into STATUS register ;set bank to original
                                         ;state
       swapf W_temp,1       ;Swap nibbles in W_TEMP and place result in W_TEMP
       swapf W_temp,0       ;Swap nibbles in W_TEMP and place result into W
       bcf   intcon,intf
       retfie
```

**Program Example 6.5: An interrupt using data calculated in the program**

Therefore, we must accept the fact that in certain program areas we will not want to accept the intrusion of an interrupt under any circumstances, with or without context saving. We call these ‘critical regions’. We can disable, or ‘mask’, the interrupts for their duration by manipulating the enable bits in the **INTCON** register. Critical regions may include:

- times when the microcontroller is simply not readied to act on the interrupt (for example during initialization – hence, only enable interrupts after initialization is complete);
- time-sensitive activities, including timing loops and multi-instruction setting of outputs;
- any calculation made up of a series of instructions where the ISR makes use of the result.

By properly applying the techniques of context saving and critical regions, we can make good use of interrupts without them displaying the more destructive side of their nature.

## 6.3 The main idea – counters and timers

### *6.3.1 The digital counter reviewed*

It is very easy to make a digital counter using flip-flops. Counters can be made which count up, count down, can be cleared back to zero, pre-loaded to a certain value, and which by the provision of an overflow output can be cascaded with other counters. A simple example is shown in Figure 6.5. Eight negative edge-triggered J–K bistables are interconnected, so that the Q-output of one drives the clock input of the next. With J and K both tied to Logic 1, the flip-flop toggles on every input negative edge. The counter holds an 8-bit binary number, made up of the eight Q-outputs of the bistables, where $Q_7$ is the most significant and $Q_0$ the least significant. It counts up by one on the negative edge of every incoming clock cycle.

The output timing diagram is shown in the lower part of the figure. It can be seen that after one input cycle $Q_0$ has gone to Logic 1. After 16 input cycles have been completed (i.e. during cycle 17) the 8-bit word forms $00010000_B$, i.e. $16_D$, and after 31 cycles it forms $00011111_B$, i.e. $31_D$. When 255 input cycles have been completed the counter holds the word $11111111_B$, or $FF_H$. If another input cycle comes along, then all flip-flops ripple through to 0 and the output returns to $00000000_B$. The negative-going edge of $Q_7$ can be used to indicate that the counter has overflowed.

The counter of Figure 6.5 can be reset to zero if the clear line is activated. With a little more complexity it is possible to add the facility to pre-load the counter with any number desired. By so doing we gain a versatile digital subsystem which becomes the basis for a microcontroller counter. This can be represented as in Figure 6.6. The only interconnections of significance are the clock input, the overflow output and the 8-bit Read or Load capability, which can be gated to share a single bi-directional data path.

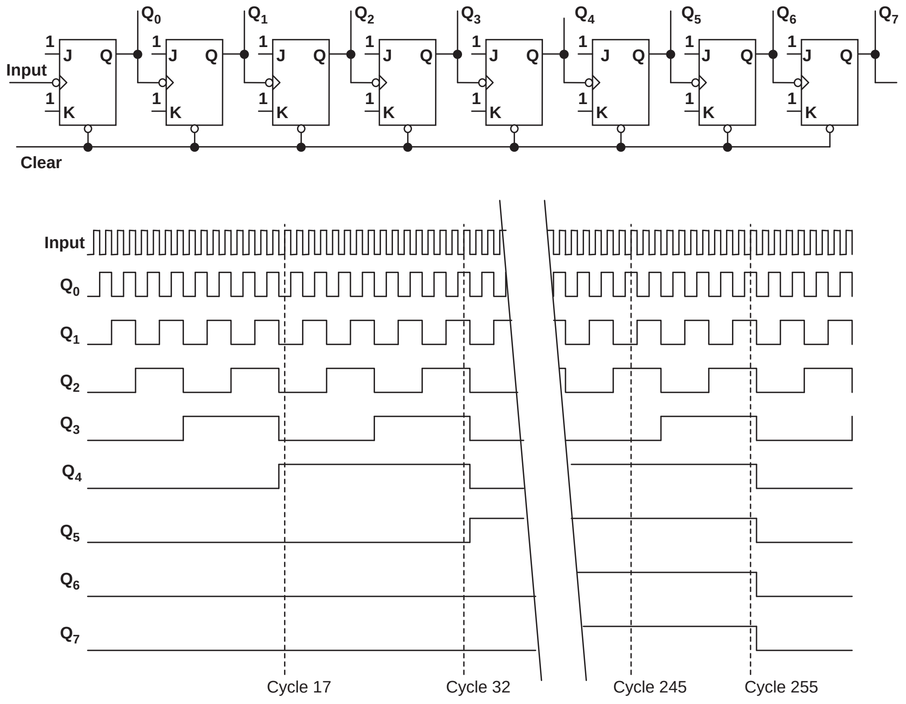


**Figure 6.5: A digital counter made of eight flip-flops**

## *6.3.2 The counter as a timer*

It is extremely useful for a microcontroller to be able to count – widgets passing on a conveyor belt, for example, coins in a slot machine, or people going through a door. It is, however, especially useful if it can measure time, and the counter allows us to do this.

Suppose the input signal of Figure 6.5 was a stable 1 kHz clock frequency. Then the counter would increment exactly every 1 ms. After 16 clock cycles, exactly 16 ms would have elapsed, after 31 cycles 31 ms and so on. By starting the clock input at a moment of choice, it is therefore possible to measure elapsed time. The resolution of the measurement is determined

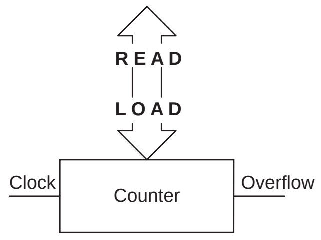


**Figure 6.6: The digital counter in block diagram form**

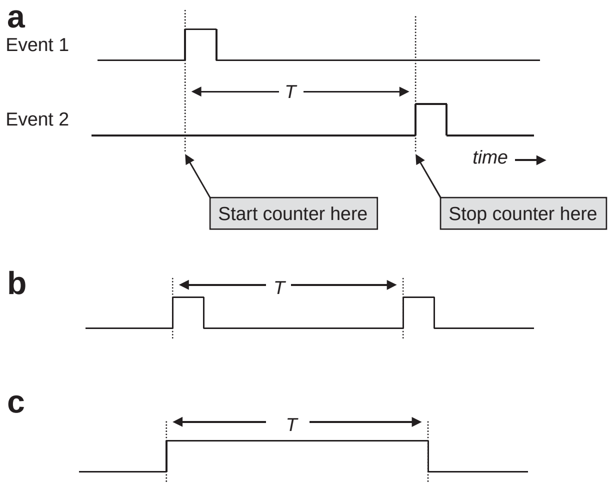


**Figure 6.7: The challenge of time measurement**

by the period of the clock. In this example the resolution is 1 ms and we can't measure anything less than that, or a fraction of it! Again, for the 1 ms input period, the 8-bit counter can measure up to 255 ms before overflowing. The use of counters as timers is so important that the counter is often called a counter/timer (C/T), or simply a timer, to reflect this importance.

An obvious application of the counter/timer is to measure the time between two 'events'. These events may both be externally generated. Alternatively, the first is generated by the microcontroller and the second happens some time later, as a response. It may also be necessary to measure the time between two pulses or the duration of a single pulse. The general requirement is illustrated in Figure 6.7. The actual measurement seems easy – start the counter/timer running when the first event occurs and stop it at the moment of the second. In practice, this poses a number of challenges. For an accurate measurement, the start and stop of the counter/timer must be perfectly synchronised with the events. The best way of doing this is by using an interrupt. If we don't have an interrupt, then we will have to continuously scan the input to detect when the event occurs – in which case it's hardly worth using the counter/timer, as we might as well do the timing in the software. If there are two external events on two different lines then we still have a problem as, with the PIC 16 Series, we only have one external interrupt.

We will see a good example of this sort of time measurement in Chapter 10. We will also see enhancements to the counter/timer that get over the problem of accurately synchronising the start and stop of the counter/timer with the events it is measuring.

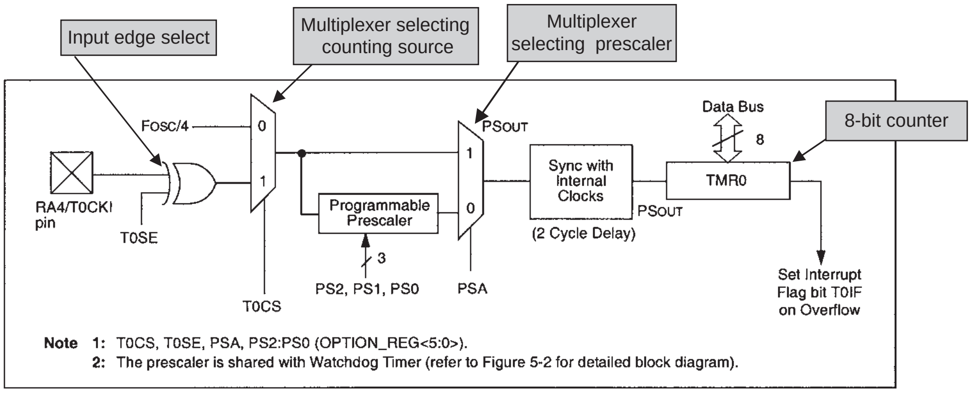

Input edge select
Multiplexer selecting counting source
Multiplexer selecting prescaler
8-bit counter
FOSC/4
RA4/T0CKI pin
T0SE
T0CS
Programmable Prescaler
PS2, PS1, PS0
3
PSA
PSOUT
Sync with Internal Clocks
(2 Cycle Delay)
PSOUT
Data Bus
8
TMR0
Set Interrupt Flag bit T0IF on Overflow
Note 1: T0CS, T0SE, PSA, PS2:PS0 (OPTION_REG<5:0>).
2: The prescaler is shared with Watchdog Timer (refer to Figure 5-2 for detailed block diagram).

**Figure 6.8: The 16F84A Timer 0 module (supplementary labels in shaded boxes added by the author)**

## *6.3.3 The 16F84A Timer 0 module*

The 16F84A Timer 0 is typical of many simple counters/timers in smaller-scale microcontrollers. It takes an 8-bit counter like the one in Figure 6.5, connects it as an SFR in the memory map and packages it with some useful extra features. Its block diagram representation is shown in Figure 6.8, with the actual 8-bit counter labelled **TMR0**. Looking back to Figure 2.5, we can see that this appears as register **TMR0** at memory location 01 in bank 0. Like all good microcontroller peripherals, Timer 0 is configurable, controlled by a number of bits that appear in the **OPTION** register, as shown in Figure 6.9.

Looking to the left of Figure 6.8, we can see that there are two possible sources of the clock input to the **TMR0** counter. One is the **RA4** pin (i.e. pin 3 of the 16F84A – see Figure 2.1). The other is the internal instruction cycle frequency, labelled Fosc/4. The selection of the input source is made by the multiplexer controlled by bit **T0CS**, which appears in the Option register. The external input path includes the option of inverting the signal with the Exclusive OR gate, the inversion being controlled by bit **T0SE**. The output of the first multiplexer branches before reaching a second multiplexer. This selects either a direct path or the path taken through a programmable prescaler. The choice is controlled by bit **PSA** of the Option register. A complication here is that the prescaler is actually shared with the Watchdog Timer (WDT), which we meet a little later in this chapter. For now we just need to recognise that if **PSA** is set to 1, then the prescaler is assigned to the WDT and the multiplexer selects the input path which avoids the prescaler. The prescaler itself is controlled by bits **PS2**, **PS1** and **PS0** of the Option register. Inspection of these bits in Figure 6.9 shows that they allow a choice of frequency divisions of the incoming clock signal. The output

| R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 |
|---|---|---|---|---|---|---|---|
| $\overline{\text{RBPU}}$ | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 |
| bit 7 | | | | | | | bit 0 |

bit 7 **$\overline{\text{RBPU}}$:** PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer 0 module

bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

| Bit Value | TMR0 Rate | WDT Rate |
|---|---|---|
| 000 | 1 : 2 | 1 : 1 |
| 001 | 1 : 4 | 1 : 2 |
| 010 | 1 : 8 | 1 : 4 |
| 011 | 1 : 16 | 1 : 8 |
| 100 | 1 : 32 | 1 : 16 |
| 101 | 1 : 64 | 1 : 32 |
| 110 | 1 : 128 | 1 : 64 |
| 111 | 1 : 256 | 1 : 128 |

**Figure 6.9: The 16F84A OPTION register**

of the second multiplexer is synchronised with the internal clock, before becoming the input to the actual counter. When the counter overflows, it sets the timer overflow flag, one of the PIC microcontroller's four interrupt sources, which we met in Figure 6.2.

## 6.4 Applying the 16F84A Timer 0, with examples using the electronic ping-pong program

A simple counter/timer like the Timer 0 can be used for many applications. We will look at two examples. Both are based on the electronic ping-pong program and can be readily simulated.

### *6.4.1 Object or event counting*

The simplest application of Timer 0 is to use it as a counter, counting pulses entering the microcontroller through the external input. Looking at the electronic ping-pong circuit (Appendix 2, Figure A2.1), we see that the right paddle is connected to Pin 3 of the 16F84A. The program of Program Example 6.6 is a very simple counting example. It enables the counter appropriately and uses the right paddle as the counter input, continuously displaying the current value on the LEDs connected to Port B.

To configure Timer 0, we'll need to select its external input, i.e. **T0CS** = 1. The input edge that we trigger from is not too important. As there is a risk of switch bounce, however, we will choose the edge associated with switch release, i.e. the rising edge, as there is less likelihood of bounce. Therefore, **T0SE** = 0. We will not want the prescaler, as we wish to count the exact number of switch presses; therefore, **PSA** = 1. Hence the values of **PS2**, **PS1** and **PS0** do not matter (as this application does not make use of the WDT). All Option register bits that have not been mentioned in this paragraph are not of importance to the ping-pong program, so will be arbitrarily set to 0. A final value for the Option register setting is thus $00101000_B$.

```
;*******************************************************************
;cntr_demo                                     Counter Demonstration
;This program demos Timer 0 as counter, using ping-pong hardware
;TJW 15.4.05                                         Tested 15.4.05
;*******************************************************************
;Clock freq 800kHz approx (RC osc.)
;Port A 4    right paddle (ip) Counter input.
;       2    "out of play" led (op)
;Port B 7-0  "play" leds (all op)
;Interrupts not used
;Config Word: RC oscillator, WDT off, PU timer on, code protect off
;
      #include p16f84A.inc
;
             org   00
; Initialise
             bsf   status,rp0   ;select memory bank 1
             movlw B'00011000'
             movwf trisa        ;port A according to above pattern
             movlw 00
             movwf trisb        ;all port B bits output
             movlw B'00101000'  ;set up TMR0 for external input, +ve edge,
                                                           ;no prescale

             movwf TMR0         ;as we are in Bank 1, this addresses OPTION
             bcf   status,rp0   ;select bank 0
;
             movlw 04     ;switch on "out of play" led to show power is on
             movwf porta
loop         movf  TMR0,0 ;Continuously display Timer 0 on Port B
             movwf portb
             goto  loop
             end
```

**Program Example 6.6: Using Timer 0 as a counter**

This program can be run on the ping-pong hardware, in which case every press of the right paddle causes a binary display on the play LEDs to increment by one.

**Programming Exercise 6.4**

Copy the program Cntr_Demo from the book's companion website into MPLAB and create a project around it. Build the project and enable the simulator. Open a Watch window, displaying **PORTB** and **TMR0**. Open the Stimulus Controller and set Pin **RA4** to Pulse high, with a pulse width of one cycle. Animate the program, 'fire' the input pulse and see how the Timer 0 and Port B SFRs count up.

Download the program to the pingpong hardware and run it, if you have the means to do this.

### *6.4.2 Hardware-generated delays*

In the original ping-pong program software-generated delays are used to time how long the LEDs are to be illuminated. This is only acceptable in simple programs, as in software-generated delays the CPU is doing nothing useful during the whole of the delay. Now that we have a counter/timer at our disposal, we can use it to generate the delay and if necessary the CPU can busy itself with other things. This seems quite simple, but a small problem presents itself: how do we know when the delay period is up? If we have to keep checking the timer value, then we will have made little progress. This is where the 'interrupt on overflow' comes into its own. If things are set up so that an interrupt is generated as the delay ends, then we have a powerful means of creating efficient delays.

As a first step, let's replace the 5 ms software delay subroutine in the ping-pong program with a delay controlled by Timer 0. The internal clock is approximately 800 kHz and the instruction cycle rate (Fosc/4) is therefore 200 kHz, or a period of 5 μs. Now with this clock frequency, Timer 0 would count up to its maximum value (255) in 255 × 5 μs, or 1275 μs, and would overflow on the next cycle, i.e. after 1280 μs. We can, however, make use of the prescaler here. If the incoming signal is divided by 4 (i.e. **PS2**, **PS1**, **PS0** set to 001), then Timer 0 will overflow after 256 × 4 × 5 μs, or 5.120 ms. This is very close to the 5 ms we're looking for, but it's not quite exact.

Although the ping-pong program does not need accurate timing, suppose we genuinely needed a delay very close to 5 ms? Let us divide the incoming clock by 8 instead of 4, which gives a divided frequency of 25 kHz, or a period of 40 μs. Now 125 Timer 0 input cycles will cause a delay of 40 × 125 μs, or 5.00 ms, which is exactly our target. If we arrange for this prescaling, and at the start of each delay pre-load Timer 0 with 256 − 125, i.e. $131_D$, then an exact delay, terminated by the interrupt on overflow, is possible.

An implementation of this approach is shown in the program sections in Program Example 6.7. This includes both the initialisation section and the revised delay subroutine. Interrupts are *not*

```
...
;Initialise
      org   0010
start bsf   status,5     ;select memory bank 1
      movlw B'00011000'
      movwf trisa        ;port A according to above pattern
      movlw 00
      movwf trisb        ;all port B bits op
      movlw B'00000010'  ;set up TMR0 for internal input, prescale by 8
      movwf TMR0         ;as we are in Bank 1, this addresses OPTION
      bcf   status,5     ;select bank 0
...
...
;introduces delay of 5ms approx
      delay5 movlw D'131'        ;preload counter, so that 125 cycles, each
                                 ;of 40us, occur before timer overflow
      movwf TMR0
del1  btfss intcon,2             ;test for Timer Overflow flag
      goto del1                  ;loop if not set
      bcf intcon,2               ;clear Timer Overflow flag
      return
```

**Program Example 6.7: Using Timer 0 in the 'delay5' subroutine**

enabled and the subroutine determines when the delay is complete by testing the overflow interrupt flag. The advantage to the programmer is that timing is now achieved by manipulating the Timer 0 settings, rather than by adjusting the software routine. The 'interrupt on overflow' has not been enabled, as it would in this instance offer little advantage. In a more demanding program, however, the interrupt could be enabled and the time spent in the delay used to undertake other CPU activities.

**Programming Exercise 6.5**

Modify the ping-pong program to include the changes given in Program Example 6.7. Using Debugger > Settings ensure that the clock frequency is set to 800 kHz. Use the Stopwatch facility to check the time duration of the new delay subroutine. How much do the **call**, **return** and Timer loading instructions add to the delay? Can you fine-tune it to improve its accuracy?

## 6.5 The Watchdog Timer

There is another timer in the 16F84A that we need to take note of, even though it is not normally used in simple applications. This is the Watchdog Timer (WDT). A big danger with any computer-based system is that the software fails in some way and that the system locks up or becomes unresponsive. In a desktop computer such a lock-up can be annoying and one would normally have to reboot. In an embedded system it can be disastrous, as there may be no user to notice that there is something wrong and maybe no user interface anyway. The WDT offers a fairly brutal 'solution' to this problem. It is a counter, internal to the microcontroller, which is continually counting up. If it ever overflows, it forces the microcontroller into Reset (Figure 2.10). It is up to the programmer to ensure that within the program the WDT is repeatedly cleared. This is done with the instruction **clrwdt**. It is only when the program ceases to run correctly that these instructions are no longer executed and the overflow occurs.

A WDT Reset is generally not good news for an embedded system, as all current settings are of course destroyed and the program starts again. It is, however, better than a program which is not running at all. Note that the WDT leaves one clue of its action behind, and that is through the **TO** bit in the Status register (Figure 2.3). It is possible to test this bit towards the beginning of a program and hence distinguish between a Power-on Reset and a WDT Reset.

The 16F84A WDT is enabled by one of the configuration bits, as seen in Figure 2.6. Thus, it either runs or it doesn't for the duration of the time the microcontroller is switched on. It is driven by an internal RC oscillator, which gives a nominal time-out period of 18 ms. This, however, is to some extent dependent on temperature, supply voltage and variation from device to device. It can be extended by applying the Timer 0 prescaler to it, in which case the time-out period can be stretched up to 128 $\times$ 18 ms, or around 2.3 seconds.

## 6.6 Sleep mode

Although we are considering timing in this chapter, it is an appropriate moment to consider one aspect of microcontroller operation when time is almost suspended – the Sleep mode. This represents an important way of saving power. The microcontroller can be put into this mode by executing the instruction **SLEEP**, seen in Appendix 1. Once in Sleep mode, the microcontroller almost goes into suspended animation. The clock oscillator is switched off, the WDT is cleared, program execution is suspended, all ports retain their current settings, and the **PD** and **TO** bits in the Status register (Figure 2.3) are cleared and set respectively. If enabled, the WDT continues running. Under these conditions, power consumption falls to a negligible amount – Ref. 2.1 quotes a typical value of 1 μA, under specific ideal operating conditions.

Once asleep, the microcontroller of course needs something to wake it up again. The 16F84A wakes from Sleep in the following situations:

- *External reset through **MCLR** pin*. While this causes a wake-up, it also resets the microcontroller; therefore, its use seems limited to complete program restarts. It *is* possible, however, to detect that the microcontroller has just been in Sleep mode, due to the state of the **PD** pin in the Status register.
- *WDT wake-up*. The function of the WDT is a little different in Sleep. Looking at Figure 2.10, it can be seen that the WDT is blocked from causing a reset when in

Sleep. Instead, on overflow it just causes a wake-up from Sleep, and the microcontroller continues program execution from the instruction following the Sleep mode.

- *Occurrence of interrupt*. As Figure 6.2 indicates, any individually enabled interrupts cause wake-up from Sleep, regardless of the state of the Global Interrupt Enable. Timer 0 cannot, however, generate an interrupt, as the internal clock is disabled.

On wake-up, the oscillator circuit is restarted. For any crystal oscillator mode this means that the $T_{OST}$ timer, seen in Figure 2.10, is also activated. It must complete its count before program execution can resume. Therefore, like a human being, the 16F84A takes a finite time to wake up and be ready for action.

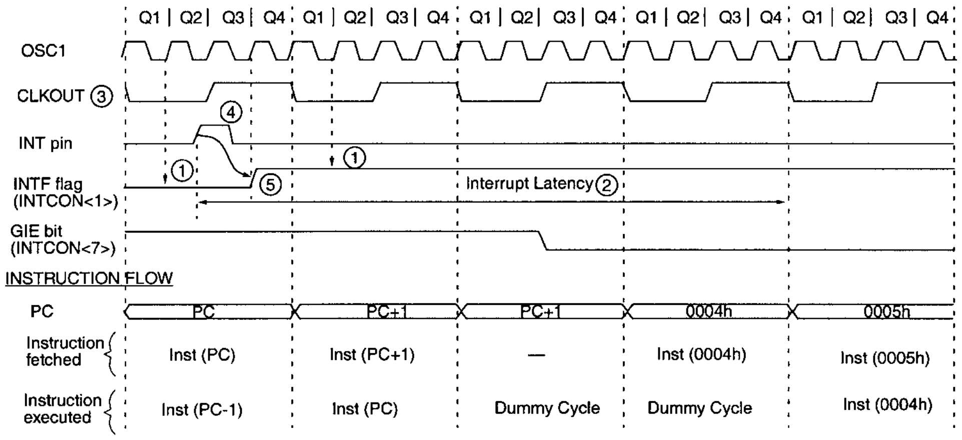
The Sleep mode is extremely powerful for products that must be designed in a power-conscious way. Many devices are not continuously active when powered. If put into Sleep when not in use, their power consumption can be dramatically reduced.

## 6.7 Taking things further – interrupt latency

The purpose of the interrupt is to attract the attention of the CPU quickly, but how quickly does this actually happen? The time between the interrupt occurring and the CPU responding to it is called the 'latency'. The latency is dependent on certain aspects of hardware and ultimately can also depend on the characteristics of the program running. The timing diagram of Figure 6.10 shows how the mid-range PIC family responds to an enabled external interrupt. The interrupt itself can be seen as a positive-going pulse on the **INT** pin line. This causes the interrupt flag **INTF** to be set. This flag is sampled on the Q1 cycle of the internal oscillator clock. Once this is done, the CPU has detected the interrupt and the sequence then follows that of Figure 6.4. Two dummy cycles are needed to save the Program Counter to the Stack, reload it with $0004_H$ and fetch the instruction at that address.

**Programming Exercise 6.6**

Working with the int_demo1 program again, set the clock frequency to 4 MHz using **Debugger > Settings**. Enable the Stopwatch (under Debugger) and single-step through the program. See how the Stopwatch updates elapsed time in a predictable way. Now instigate the interrupt. Notice how the Stopwatch records a latency of two instruction cycles. At the end of the ISR see that the **retfie** instruction also takes two cycles.



Note 1: INTF flag is sampled here (every Q1).
2: Interrupt latency = 3-4 TCY where TCY = instruction cycle time.
Latency is the same whether Instruction (PC) is a single cycle or a 2-cycle instruction.
3: CLKOUT is available only in RC oscillator mode.
4: For minimum width of INT pulse, refer to AC specs.
5: INTF is enabled to be set anytime during the Q4-Q1 cycles.

**Figure 6.10: 16F84A external interrupt latency**

## Summary

- Interrupts and counters/timers are important hardware features of almost all microcontrollers.
- They both carry a number of important hardware and software concepts, which must be understood.
- The basic techniques of using interrupts and counters/timers have been introduced in this chapter. There is considerably increased sophistication in their use in more advanced applications.

## Questions and exercises

Try to complete Programming Exercises 6.1 to 6.6 and the questions below.

1. (a) The **INTCON** register in a certain 16F84A application is found to read 10110000. What interrupts are enabled?
   (b) As the program executes, it is found to read 00110010, 00110000, 10110000 in sequence. Explain the likely cause of each of these changes.

2. An inexperienced programmer has written the code shown below for a 16F84A microcontroller, where the interrupt source is Timer 0 overflow. Identify all errors and rewrite the program fragment correctly.

```
       org    0000
       goto   start
       org    0014
       goto   my_interrupt
;Program starts here
start bsf status,5
...
;this is the interrupt routine
my_interrupt movlw   0f
       addwf  counter1,1
       btfsc  flags,2      ;test motor run flag
       clrf   overflow
       return
...
```

3. The Timer 0 module of a 16F84A is initiated by loading the **OPTION** register with value 00000011. The oscillator frequency is 8 MHz.
   (a) Under these conditions, what is the frequency at the input to the Timer?
   (b) If the counter is initially cleared to zero, how long does it take before it first overflows?
   (c) If **INTCON** was initially cleared to 0, what is its value immediately after this overflow occurs?

4. A machine counts envelopes which are being packaged in packs of 150. The machine is controlled by a PIC 16F84A. A sensor connected to the **RA4/T0CK1** pin produces a logic pulse every time an envelope passes it.
   (a) Describe how you would configure the Timer 0 to count the envelopes. Indicate what value you would set in the **OPTION** register.
   (b) Explain what strategy could be used to allow the microcontroller program to detect when the number 150 had been reached.

5. In an application using the 16F84A, a regular timed interrupt is required. The clock oscillator frequency is 4 MHz and an interrupt frequency in the region of every 2 ms is required. Describe how you would configure the Timer 0 module.

6. The following program fragment is for a 16F84A-based system, with a clock frequency of 10 MHz. Explain in as much detail as possible how the counter/timer and interrupts are being used.

```
         bsf     status,5    ;select register bank 1
         movlw   07
         movwf   option
         bcf     status,5    ;select bank 0 for later
                                 ;memory transfers
         bcf     intcon,2    ;
         bsf     intcon,5    ;
         bsf     intcon,7    ;
wait     goto    wait        ;wait for next Interrupt
```

7. An application requires use of the WDT, with timeout period around 150 ms. Describe all settings which must be used in order to achieve this. If a timeout does occur, how can this be detected by the program?