Lesson 13
The TMR0 Register

Overview

Introduction

The PIC16F84A includes a timer register, TMR0. This register can, among other things, be used to manage performing multiple tasks simultaneously.

In this section

Following is a list of topics in this section:

The TMR0 Register

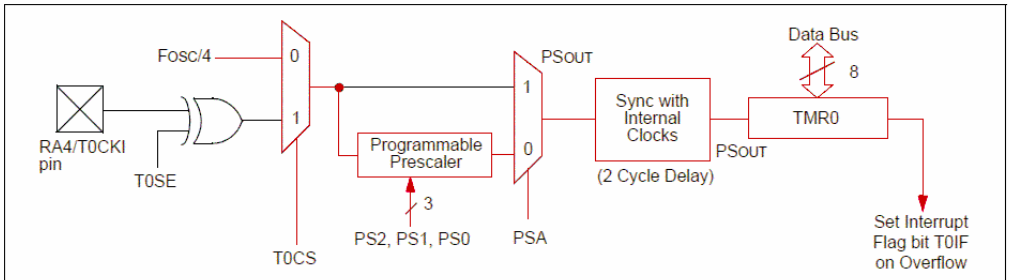
Introduction

The TMR0 (timer 0) register, as its name implies, can be used to measure elapsed time. The time base for the register can be selected to be either the processor clock or an external clock. Associated with the timer is a prescaler, which can adjust the resolution of the timer register. The timer can be read, and will set a bit (and optionally an interrupt) when the register overflows.

TMR0 Structure

The timer is controlled by a number of bits in the Option Register. All processors are nothing more than a collection of gates. While this may be hard to tell in a very complex processor like a Pentium, the PIC is a very simple processor, and sometimes this actual simplicity makes itself obvious. The timer is one of those cases.

On page 19 of your PIC16F84A datasheet, you will see the following block diagram:



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).

For this lesson, we will only concern ourselves with the red path through these gates. All of the acronyms along the bottom of the diagram refer to bits in the option register, except for T0IF (Timer 0 interrupt flag), which is a bit in the INTCON register.

Starting at the left, the processor clock is divided by four and fed into a gate. This division by four results in a single cycle per instruction execution. In the case of a 4 MHz processor crystal, this conveniently results in a 1 MHz clock.

When bit T0CS (Timer 0 clock select) is false, the clock is fed into a prescaler. The prescaler ratio is set by bits PS2, PS1 and PS0. If the PSA bit is false, the output of the prescaler is then routed to the TMR0 register, after synching with other internal clocks. This causes a 2 cycle delay, which is only apparent if we load the TMR0 register with a value.

Every cycle of the prescaler output increments TMR0 by one. We can read as well as write the contents of TMR0. Whenever TMR0 overflows, the T0IF bit in the INTCON register is set.

If the INTCON bit T0IE (Timer 0 interrupt enable) is set, this transition of the T0IF bit causes an interrupt. We will not discuss interrupts this lesson, so for now, we will always take care to clear T0IE.

The Option Register

Introduction

The option register is used to control a number of processor features. The least significant six bits of the option register control the timer logic that was examined earlier.

Option Register bits

In the datasheet, the following drawing is on page 11:

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

The meaning of those bits is as follows:

bit	name	purpose
0-2	**PS0-PS2**	These three bits together determine the division ratio of the prescaler.
3	**PSA**	This bit determines whether the prescaler will be used for the TMR0 register or for the watchdog timer
4	T0SE	This bit determines whether the rising or falling edge will trigger a transition when RA4 is used as the input to TMR0
5	**T0CS**	This bit determines whether the processor clock or RA4 will be used as the input to TMR0
6	INTEDG	The PIC can be programmed such that a transition on RB0 causes an interrupt. This bit determines whether that interrupt occurs on the leading or trailing edge
7	RBPU	Clearing this bit enables weak pull-up resistors on all the PORTB inputs. For low current applications like reading switches, this can eliminate the need for external pull-up resistors.

In this lesson we will not use an external input to the clock, so only those bits which are in bold will be used.

Using the Timer

Introduction

The combination of the timer, prescaler, and interrupt bits means that there are a number of steps that need to be taken in order to use the timer effectively.

Selecting the parameters

In order to set up the timer, it is necessary to first decide the time interval needed. The basic timer rate is one microsecond (with a 4 MHz crystal). This one microsecond clock is divided by the prescaler, which can be set to divide by 2, 4, 8 16, 32, 64, 128 or 256. The timer register itself has 8 bits, so it can count to 256. Thus, it is necessary to service the timer with software at least every 256*256 microseconds, or 65.536 milliseconds (assuming a 4 MHz clock).

The timer register itself can be used to divide by any arbitrary number by simply reloading it whenever the register overflows, and additional software counters can be updated based on the timer, so it is possible to arrange any desired time. The catch is that, the higher the resolution needed, the more frequently software must service its counters.

Consider for a moment an application that requires a 10-millisecond timer. If the prescaler is set to divide by 64, the timer register can count to 16.384 milliseconds. If the timer register is preloaded with 100, then the timer will expire in 9.984 milliseconds.

If one were to use a prescaler division of 16, it is possible to get a delay of exactly 10 milliseconds, however, it would require servicing the timer in software every two milliseconds and maintaining a counter in software. The programmer needs to consider how good is “good enough”, balanced against the complexity and the potential that time could be taken from other tasks to watch the clock.

An alternative is to use the timer to count off 9.984 milliseconds, and then use another approach, perhaps simply looping, to count the additional 16 microseconds.

In some applications where timing is critical, the designer will often select the crystal frequency to allow for the exact time schedules demanded by the application.

Setting up the timer

To set up the timer, one must first disable interrupts so that an interrupt doesn’t occur when the timer expires. Then, enable the timer and assign the prescaler to the timer. Establish the prescaler value, and finally, load the timer register.

The bits for enabling the timer and assigning the prescaler to the timer, as well as the bits that set the prescaler division ratio are all in the same register. Thus, these values may be set bit by bit, or by simply loading the Option register with a value (assuming it is possible to determine benign values for the other bits).

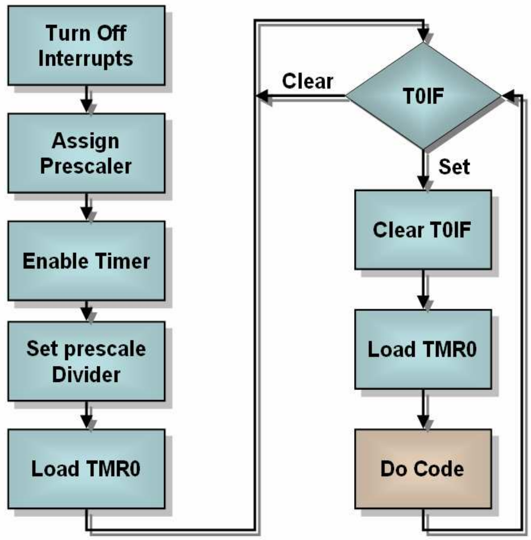
Whenever the timer expires, the T0IF bit in the INTCON register will be set. We must clear this bit, reload the timer register, and then execute the code that is to be done at this time.

Continued on next page

Using the Timer, Continued

Setting up the timer (continued)

Thus, the process looks something like:



In code, the setup portion might look something like:

```
banksel     INTCON
bcf         INTCON,T0IE         ; Mask timer interrupt

banksel     OPTION_REG
bcf         OPTION_REG,T0CS     ; Enable timer
bcf         OPTION_REG,PSA      ; Prescaler to timer
bcf         OPTION_REG,PS2      ; \
bsf         OPTION_REG,PS1      ;  >- 1:16 prescale
bsf         OPTION_REG,PS0      ; /
movlw       D'100'              ; Timer will count
movwf       TMR0                ; 156 (256-100) counts
```

Clearly, the individual bits in the option register could all be set with a single store. If we didn't care about the RB0 interrupt, the weak pullups, or the transition of RA4, then instead of five bit manipulations we could have said:

```
movlw       B'10000011'         ; Set up prescaler and
movwf       OPTION_REG          ; timer
```

The execution loop might look something like:

```
main
        btfss       INTCON,T0IF   ; Did timer overflow?
        goto        main          ; No, hang around some more
        movlw       D'100'        ; Timer will count
        movwf       TMR0          ; 156 (256-100) counts
        bcf         INTCON,T0IF   ; reset overflow flag
        call        DoCode        ; Execute main code
        goto        main          ; Go back and wait
```

Multitasking

Introduction

When a processor is applied to an embedded system of any sort, there are typically several things that need to be managed independently. The processor is expected to do several things at once. Few real processors are capable of this feat, so the developer is faced with making the processor appear to be doing several things at once. The core of this job is the task scheduler.

Types of Schedulers

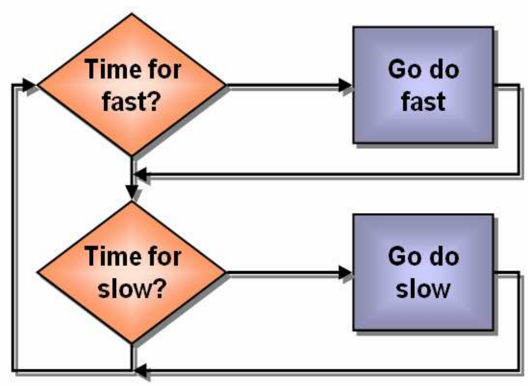
In most real time executives, the task scheduler is interrupt driven. This allows higher priority tasks to interrupt lower priority tasks. This can be a fairly complex business and it is fraught with problems. In particular, the unpredictability of what things may be interrupted when means that some types of interactions are very difficult to test.

For this reason, high reliability systems often use a more deterministic scheduler. Basically, the scheduler keeps track of the time and dispatches each task when it is time for that task to run. This is far simpler than an interrupt driven scheduler, but it does have one big downside; *every* task must be guaranteed to complete within the time allotted for the fastest task.

While this may sound like a significant limitation, in most PIC applications it simply means that any looping or waiting for I/O devices must be avoided within the tasks themselves. Even though the PIC is not a terribly fast processor, it is unusual for an embedded application to involve computation that actually takes significant time.

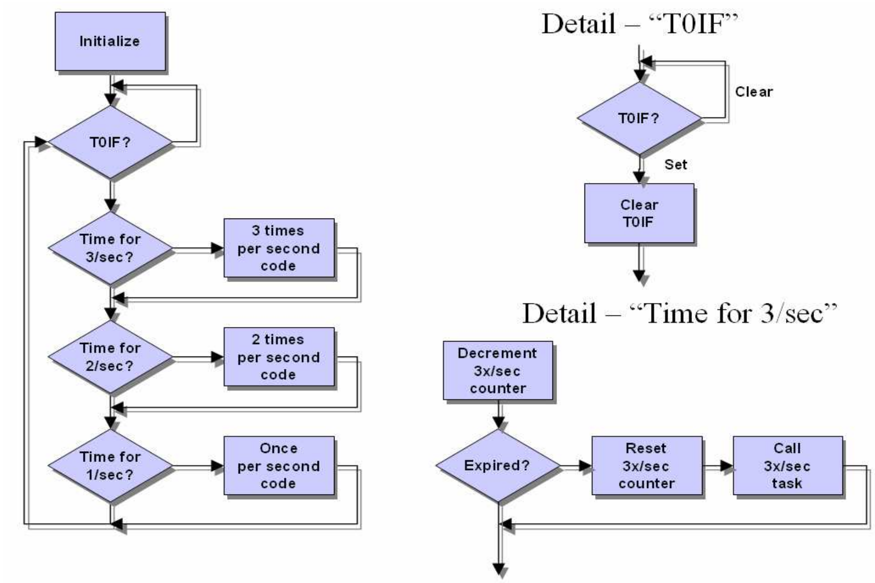
Design of the scheduler

The scheduler itself can be fairly simple. All it takes is to loop while watching the clock, then do the appropriate task when its time is due:



This is accomplished by setting the timer to some small value and maintaining a counter for each task. Ideally, all inputs would occur at the beginning of one of the task time frames, and output at the end. In practice, it is often necessary to do I/O at the beginning and end of multiple time frames. This can lead to difficult to diagnose interactions, so one must be alert to these possibilities in the external circuitry.

Watch the blinkenlights

Introduction	For our first example, let's flash the LEDs yet again. This time, we will flash each of the PIC-EL's LEDs at independent rates. As a simple example, let's choose one, two and three times per second.
Planning the Application	To start, the experienced developer will sketch out how the application is intended to work. We can use a similar scheme to what was presented earlier, but add a third task:  It is also necessary to select the prescaler value to be used to set the time frames. Since the longest time we have is one second, and we would like our counters to stay within one byte, we need to cause the timer to expire no more quickly than 256 times per second, or once every 3.9 milliseconds. If we allow the timer to run it's full, 256 cycle course, and set the prescaler to divide by 16, then the T0IF bit will be set every 4.096 milliseconds (with a 4 MHz processor clock).
The individual tasks	Our three tasks are each quite simple, and all the same. Each needs only to complement the state of the LED to which it is assigned, and then call a routine to send the outputs.
Initialization	For initialization, we need to initialize the timer, preload each of the three counters, set the initial state of the LEDs, and set the three LED bits of PORTB to be outputs.

Continued on next page

Watch the blinkenlights, Continued

Scheduler Code

The code for the scheduler looks much like the examples presented above, except that we have three tasks to manage:

```
;=====================================================================
; Main program loop here
;=====================================================================
main
               btfss   INTCON,T0IF    ; Did timer overflow?
               goto    main           ; No, hang around some more
               bcf     INTCON,T0IF    ; reset overflow flag

;---------------------------------------------------------------------
;       Check for three times per second
;---------------------------------------------------------------------
               decfsz  Hz3Cnt,F       ; Count down until Hz3
               goto    $+4            ; Not time yet
               movlw   HZ3TIME        ; Reset the counter so
               movwf   Hz3Cnt         ; it's available next time
               call    Hz3            ; Go do thrice per second code

;---------------------------------------------------------------------
;       Check for two times per second
;---------------------------------------------------------------------
               decfsz  Hz2Cnt,F       ; Count down until Hz2
               goto    $+4            ; Not time yet
               movlw   HZ2TIME        ; Reset the counter so
               movwf   Hz2Cnt         ; it's available next time
               call    Hz2            ; Go do twice per second code

;---------------------------------------------------------------------
;       Check for once per second
;---------------------------------------------------------------------
               decfsz  Hz1Cnt,F       ; Count down until Hz1
               goto    $+4            ; Not time yet
               movlw   HZ1TIME        ; Reset the counter so
               movwf   Hz1Cnt         ; it's available next time
               call    Hz1            ; Go do once per second code

               goto    main
```

Notice that we used relative jumps to avoid cluttering the loop with labels. This does have the unfortunate side effect that changes can lead to surprising results when code is added, so the developer may prefer to label each section instead.

Task code

Each of the three tasks are very simple, and almost identical:

```
;---------------------------------------------------------------------
;       Three times per second code
;---------------------------------------------------------------------
Hz3
               movlw   LED3M          ; Toggle LED3 bit by
               xorwf   Outputs,F      ; XORing with current state
               call    SendOut        ; Set outputs
               return
```

Notice how we can toggle a bit by XORing it.

Continued on next page

Watch the blinkenlights, Continued

Output Routine

The output routine is similarly simple:

```
;=====================================================================
;  Subroutines
;=====================================================================
SendOut
               movf    Outputs,W      ; Pick up the output word
               movwf   PORTB          ; And send it to the world
               return
```

Initialization

For the initialization, we have a number of steps.

First, we set up the timer:

```
;=====================================================================
;  Mailine begins here -- Initialization
;=====================================================================
start

;---------------------------------------------------------------------
;       Set up timer
;---------------------------------------------------------------------
               errorlevel     -302
               banksel        INTCON
               bcf            INTCON,T0IE           ; Mask timer interrupt
       ; Normally, we would have simply loaded a constant, but
       ; the code below makes it explicit what we are doing
               banksel        OPTION_REG
               bcf            OPTION_REG,T0CS       ; Enable timer
               bcf            OPTION_REG,PSA ; Prescaler to timer
               bcf            OPTION_REG,PS2 ; \
               bsf            OPTION_REG,PS1 ;  >- 1:16 prescale
               bsf            OPTION_REG,PS0 ; /
```

Then the I/O ports:

```
;---------------------------------------------------------------------
;       Set up I/O
;---------------------------------------------------------------------
               banksel        TRISB          ;
               clrw                          ; Make all PORTB bits output
               movwf          TRISB          ;
               banksel        PORTA          ; Back to bank 0
               errorlevel     +302
```

And finally, the memory locations:

```
;---------------------------------------------------------------------
;       Initialize memory
;---------------------------------------------------------------------
               movlw          B'00001110'    ; Initially set all LEDs
               movwf          Outputs        ; to off
               movlw          HZ1TIME        ; Initialize the counters
               movwf          Hz1Cnt         ; for the three time domains
               movlw          HZ2TIME
               movwf          Hz2Cnt
               movlw          HZ3TIME
               movwf          Hz3Cnt
```

Continued on next page

Watch the blinkenlights, Continued

Variables and constants

We need to set up six manifest constants, and four file register locations. Three of the constants are masks which contain a '1' bit in a position corresponding to the LED bit in PORTB.

The other three initialize the counters we will use for the three time domains. The once per second timer must be set to 1000 ms / 4.096 ms = 244. The others are set to 500 / 4.096 = 122 and 333 / 4.096 = 81. These values are not exact. If we wish precise times we would need to maintain multiple byte counter and a smaller prescaler setting, or select a specific crystal frequency for the application.

```
;=====================================================================
;       Manifest Constants
;=====================================================================
LED1M           equ     B'0001000'      ; Mask for LED1
LED2M           equ     B'0000100'      ; Mask for LED2
LED3M           equ     B'0000010'      ; Mask for LED3
HZ1TIME         equ     D'244'          ; Clock ticks for 1/sec
HZ2TIME         equ     D'122'          ; Clock ticks for 2/sec
HZ3TIME         equ     D'81'           ; Clock ticks for 3/sec

;=====================================================================
;       File register use
;=====================================================================
                cblock          H'20'
                        Hz1Cnt          ; Once per second counter
                        Hz2Cnt          ; Twice per second counter
                        Hz3Cnt          ; Thrice per second counter
                        Outputs         ; Output storage
                endc
```

Notice that if we "fudge" the three counters and make them 240, 120, and 80 the three LEDs will come into synch periodically.

Running and testing the program

The complete source code for the application is available on the web site. Besides what is listed here, the file contains the same starting directives we always use, a goto to skip around the subroutines, and an end statement.

The interested experimenter might try different frequencies of the LEDs. With the same prescaler setting and logic, the LEDs cannot be flashed much slower than once per second, but they can be sped up until the flashing is barely visible.

State Variables

Introduction

The previous example program managed multiple tasks with different time frames, but the application itself wasn't very interesting. More interesting behavior requires that the individual tasks be able to remember what state they are in. Statefulness is, appropriately enough, managed by state variables.

Complexity of state variables

A task may need to remember a small number of states, or it may have quite a rich collection of states to track. Our state variable, then, may be a single bit, or it may be a complex combination of values.

In addition, the state variable(s) may be private to the task, or used to coordinate the actions of several tasks. It is important to understand these uses and keep the purpose of a particular state variable well focused. When tasks become more involved, and there are a larger number of them, interactions can become very difficult to diagnose unless these interactions are well controlled.

One common use of state variables is to break a lengthy operation into multiple pieces. For example, if we are managing a keyer and want to display something on an LCD, the LCD operation could take enough time to make the keyer operation a little rough. A state variable might allow the LCD task to send one letter at a time to the LCD, and relinquish the processor so that the paddle can be checked between letters.

An Example Program

For our state variable example, we will build on the previous program. Instead of blinking three LEDs, we will blink only two. However, we will blink only one at a time. When the user presses a pushbutton, we will change which LED is blinking. Of course, the two LEDs will blink at different rates.

For the LED routines, we will use a single bit to remember the state. When the bit is set, (1) we will blink one LED, when clear, (0) we will blink the other.

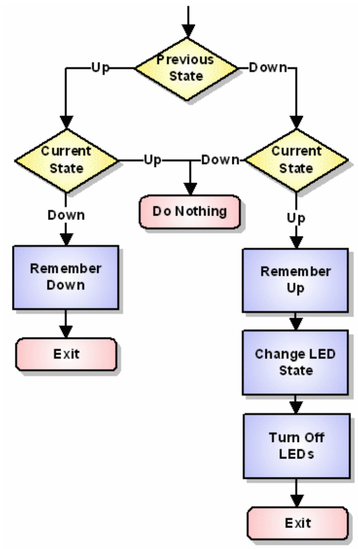
When we think through the problem, however, we might notice that the pushbutton is a bit more of a challenge. If we simply toggle the state bit when the pushbutton is down, we are likely to toggle the bit hundreds of thousands of times while the user has the button pressed. We could reduce the number of toggles by slowing the frequency at which we sample the pushbutton, but then we risk missing a quick pushbutton press. Clearly, the button will take some thought.

Continued on next page

State Variables, Continued

The Pushbutton Logic

To make the pushbutton logic make sense, we need to pay attention only to one edge of the transition. We will choose to change the program's state when the pushbutton is released.



If we sample the pushbutton at some rate, say, 20 times per second, and remember the button's state, we can then compare the previous state to the current state. When the state changes, we remember the new state. If the state changed, and it is now up, we toggle the LED state. We also want to remember to turn off the LEDs so that the "old" LED isn't left on.

```
;---------------------------------------------------------------------
;       Twenty times per second code
;---------------------------------------------------------------------
HzN
        ; Get inputs
        movf            PORTA,W
        movwf           Inputs

        ; Check button state
        btfss           PBstate,PB1     ; Was button down?
        goto            wasDown         ; Yes
wasUp
        btfsc           Inputs,PB1      ; Is button still up?
        return                          ; Was up and still up, do nothing
        bcf             PBstate,PB1     ; Was up, remember now down
        return
wasDown
        btfss           Inputs,PB1      ; If it is still down
        return                          ; Was down, still down, do nothing
        bsf             PBstate,PB1     ; remember released

        ; Button was down and now it's up,
        ; we need to flip LEDstate
        movlw           H'01'           ; Toggle LSB of LED
        xorwf           LEDstate,F      ; state
        movlw           B'00001110'     ; Turn off all LEDs
        movwf           Outputs
        return
```

Continued on next page

State Variables, Continued

The Pushbutton Logic (continued)

Notice that by selecting a single bit we can use bit test and clear instructions which tend to be a little simpler than loading and storing values and testing the status register.

The LED tasks

The LED tasks are similar to the previous example, except that prior to toggling the LED, we will test whether this LED is the active one:

```
;---------------------------------------------------------------------
;       Five times per second code
;---------------------------------------------------------------------
Hz5
        ; Check whether we are doing this
        btfss           LEDstate,0      ; Is LEDstate:0 = 0?
        return                          ; Yes, return

        movlw           LED1M           ; Toggle LED1 state
        xorwf           Outputs,F       ;
        call            SendOut         ; Set outputs
        return
```

The two times per second task is the same, except in the first instruction, we skip on the bit clear instead of set, and we load the LED 2 mask instead of the LED 1 mask.

The scheduler

The scheduler is the same as in the previous example program, except that we have chosen 2, 5, and 20 times per second. The once per second time was a little slow. We really don't change anything substantive here, only the labels and comments change to reflect the new time frames.

Constants and File Register

We no longer need the LED 3 mask since we aren't using LED 3. We need to recalculate the constants for our counters, and rename them to reflect the new frequencies.

For file register locations, we need a place to store the bit that remembers which LED we are blinking, and add a location to remember the previous pushbutton state. We also need a location for the inputs:

```
;=====================================================================
;       File register use
;=====================================================================
        cblock          H'20'
                Hz2Cnt          ; Twice per second counter
                Hz5Cnt          ; 5 times per second counter
                HzNCnt          ; 20 times per second counter
                Outputs         ; Output storage
                Inputs          ; Input storage
                PBstate         ; What state is PB?
                LEDstate        ; Which LED flashing?
        Endc
```

Input and Output

We can borrow the output routine from the previous example. The only input is the pushbutton, which we read in our 20 times per second routine shown above. If we intended to expand on this application, it may have been preferable to split the input routine into its own subroutine as we did with the output.

Continued on next page

State Variables, Continued

Testing the application	The user may download the complete source from the web page, but it may be preferable to take Lesson13a.asm and modify it as outlined above. Check that the timing on the pushbutton read is appropriate. Can the pushbutton be fooled by pressing it too quickly? Does the behavior seem "natural"?
Expanding the application	The student may wish to expand the application to include all three LEDs. This will require more than a single bit for the state variable. There are a number of possibilities here. A state value of 1, 2, or 3 could be selected. Alternatively, a bit could be assigned to each LED. Each of these choices has its price. Are there other approaches that should be considered?

PIC-EL Roulette

Introduction

As a final example, let's make a roulette wheel. Well, OK, we have some limitations. With only 3 LEDs, the wheel can only come up with 3 positions for the ball, but at least we can demonstrate the concept.

The tasks

Consider three independent tasks. One task moves the ball to the next slot. Another task provides the "friction", slowing the first task at some rate. The third task will read a button, and as long as the button is pressed, hold the first task at the maximum rate. If the fastest rate for the wheel is very fast, the user will not be able to tell the ball position when the button is released, so the final resting place for the ball will be random.

Scheduler

The scheduler will be almost exactly the same as the previous examples. The one difference is that for the 'spin the wheel' task, we will load the counter with the contents of a file register cell, rather than a constant. We also need to deal with the little detail of the ball finally stopping:

```
;=====================================================================
; Main program loop here
;=====================================================================
main
        btfss           INTCON,T0IF     ; Did timer overflow?
        goto            main            ; No, hang around some more
        bcf             INTCON,T0IF     ; reset overflow flag

;---------------------------------------------------------------------
;       Check for eighty times per second
;---------------------------------------------------------------------
        decfsz          Hz8Cnt,F        ; Count down until Hz8
        goto            $+4             ; Not time yet
        movlw           HZ8TIME         ; Reset the counter so
        movwf           Hz8Cnt          ; it's available next time
        call            Hz8             ; Go do 80X per second code

;---------------------------------------------------------------------
;       Check for twenty times per second
;---------------------------------------------------------------------
        decfsz          Hz2Cnt,F        ; Count down until Hz2
        goto            $+4             ; Not time yet
        movlw           HZ2TIME         ; Reset the counter so
        movwf           Hz2Cnt          ; it's available next time
        call            Hz2             ; Go do 20X per second code

;---------------------------------------------------------------------
;       Check for variable times per second
;---------------------------------------------------------------------
        ; Special case, if LEDrate = 0xff quit doing this
        movf            LEDrate,W       ; Pick up rate, if it's
        xorlw           H'ff'           ; ff we want to not run
        btfsc           STATUS,Z        ; this time domain
        goto            main

        decfsz          HzVCnt,F        ; Count down until HzV
        goto            $+4             ; Not time yet
        movf            LEDrate,W       ; Reset the counter so
        movwf           HzVCnt          ; it's available next time
        call            HzV             ; N times per second code
```

Continued on next page

PIC-EL Roulette, Continued

Spin the Wheel

The “Spin the Wheel” task needs to rotate an illuminated LED through the three available positions. Since the LEDs light when the corresponding PORTB pin is low, the routine must either take care to clear unneeded ‘1’ bits off the left and add them in on the right, or complement the result before storing it to PORTB. In this example, the second alternative was chosen:

```
;---------------------------------------------------------------------
;       Variable times per second code
;---------------------------------------------------------------------
HzV
                rlf             LEDstate,F      ; Move the 1 over a bit
                btfss           LEDstate,4      ; Did it roll off the end?
                goto            SetLEDs         ; No, continue on
                movlw           B'00000010'     ; Yes, reset to bit 1 on
                movwf           LEDstate        ; and store it away
SetLEDs
                movlw           B'1110001'      ; Initially turn on LEDs
                andwf           Outputs,F       ; (overkill since no other IO)
                movf            LEDstate,W      ; Pick up LED state
                xorlw           H'0e'           ; Flip because active low
                iorwf           Outputs,F       ; Set it in the outputs
                call            SendOut         ; Go do output
                return
```

Handle the button

In this application, the button routine needs to merely maintain the wheel speed at its maximum as long as the button is held down. Keeping track of the state of the button is not necessary, as it was in the previous example:

```
;---------------------------------------------------------------------
;       80 times per second code
;---------------------------------------------------------------------
Hz8
        ; Get inputs
                movf            PORTA,W
                movwf           Inputs

        ; Check button state
                btfsc           Inputs,PB1      ; Is button up?
                return                          ; Button up, do nothing
        ; Button is down
                movlw           HZVMAX          ; Set rate to
                movwf           LEDrate         ; fastest flashing
                return
```

Provide Friction

In order to make the wheel slow at some reasonable rate, we need a routine to slowly increase the counter for the ‘N’ times per second routine:

```
;---------------------------------------------------------------------
;       20 times per second code
;---------------------------------------------------------------------
Hz2
        ; Check whether rate already slowest
                movf            LEDrate,W       ; Pick up rate and xor with
                xorlw           0xff            ; ff so Z set if equal
                btfsc           STATUS,Z        ; Is rate slowest?
                return                          ; Yes, do nothing
        ; Make LEDs slower
                incf            LEDrate,F       ; Bump it down by one
                return
```

Notice that the speed at which the wheel slows can be easily adjusted by changing the frequency with which this routine is executed.

Continued on next page

PIC-EL Roulette, Continued

Constants and File Register

In this example, some additional file register locations are required, along with some slightly different constants:

```
;=====================================================================
;       Manifest Constants
;=====================================================================
PB1      equ                    H'04'   ; PORTA pin for PB1
HZ2TIME equ                     D'49'   ; Clock ticks for 20/sec
HZ8TIME equ                     D'12'   ; Clock ticks for 80/sec
HZVMAX  equ                     D'10'   ; Max rate for n/sec

;=====================================================================
;       File register use
;=====================================================================
                cblock          H'20'
                        Hz2Cnt          ; 20 times per second counter
                        HzVCnt          ; variable times per second counter
                        Hz8Cnt          ; 80 times per second counter
                        Outputs         ; Output storage
                        Inputs          ; Input storage
                        LEDstate        ; Which LED on?
                        LEDrate         ; Rate of flashing
                endc
```

Initialization

The PIC-EL Roulette program requires some slightly different initialization than the previous examples. Because fairly fast execution of the tasks is needed, the prescaler is set to divide by 4, resulting in the TMR0 register overflowing about once every millisecond (with a 4 MHz processor clock):

```
;---------------------------------------------------------------------
;       Set up timer
;---------------------------------------------------------------------
                errorlevel      -302
                banksel         INTCON
                bcf             INTCON,T0IE     ; Mask timer interrupt
        ; Normally, we would have simply loaded a constant, but the
        ; code below makes it explicit what we are doing
                banksel         OPTION_REG
                bcf             OPTION_REG,T0CS; Select timer
                bcf             OPTION_REG,PSA  ; Prescaler to timer
                bcf             OPTION_REG,PS2  ; \
                bcf             OPTION_REG,PS1  ;  >- 1:4 prescale
                bsf             OPTION_REG,PS0  ; /
```

The initial LED indication needs to be set so that when the wheel is rotated, there is a ball in there to rotate! As before, the counters must be initialized, and the LEDs are initially set all off:

```
;---------------------------------------------------------------------
;       Initialize memory
;---------------------------------------------------------------------
                movlw           B'00001110'     ; Initially set all LEDs
                movwf           Outputs         ; to off
                movwf           PORTB

                movlw           HZ2TIME         ; Initialize 20 times
                movwf           Hz2Cnt          ; per second counter
                movlw           HZ8TIME         ; and eighty times per
                movwf           Hz8Cnt          ; second counter

                movlw           B'00000010'     ; Initialize the LED
                movwf           LEDstate        ; states
                movlw           H'd0'           ; and the speed of LED
                movwf           LEDrate         ; movement
```

Continued on next page

PIC-EL Roulette, Continued

Testing the program	In the earlier discussion, uninteresting bits of code, as well as code identical to earlier example, has been left out. The student following along will find the need to fill in the blanks. On power up, the 'ball' will rotate a few steps before stopping. Thereafter, the LEDs will flash very fast as long as PB1 is held down. When the button is released, the flashing slows, and eventually stops.
Additional Experiments	The linear slowdown of the ball seems a little unnatural. The student could experiment with different approaches to come up with a more realistic behavior. We have a speaker, wouldn't it be nice to hear the metal ball clacking around?

Wrap Up

Summary	In this lesson, we have explored the timer register. We have seen how the timer can be exploited to build a simple multitasking executive, and we have written a few examples that demonstrate how multiple threads of execution can be managed.
Coming Up	In the next lesson, the use of tables to simplify PIC applications will be explored.