Using Polled or Interrupt-driven Timers

1. Overview

The purpose of this document is to show two examples of how to use the timer modules of the Microchip PIC18F8520 microcontroller. This information is applicable to both the 2004 EDU Robot Controller and the 2004 FRC Robot Controller. This document will also provide hints and a useful example of how to use the low-priority interrupts which are available to the user.

The PIC18F8520 has five timer modules, and all of them are available to the user through custom code. These timers provide a lot of power and flexibility, but require careful planning and configuration to use them effectively. There are many possible uses for the timers, but some common examples are to execute a section of code for a specified time or at specified intervals, to measure the duration of certain events, or to keep track of time elapsed. In order to accomplish these tasks, there are two methods for using the timers: polling them or using interrupts.

The easiest method to understand is polling. The polled method involves checking (or polling) a register or a bit periodically to see if a certain condition has occurred. For example, every execution loop of the program a routine can check to see if the timer has reached a certain value, or it can check to see if the timer register has overflowed (rolled over and started at zero again).

The more powerful and accurate method is to use interrupts. The interrupt-driven method allows events to be triggered automatically whenever a specified condition occurs, no matter what section of the code is being executed at the time of the condition.

2. Timer Basics

For details on the PIC18F8520 timer modules and operation you should refer to page 131 of the PIC18FXX20 Data Sheet from Microchip. Here we will summarize the more relevant points.

For all calculations you should remember that the internal clock (Fosc/4) is 10 MHz. Therefore each tick of a timer is 10^{-7}seconds. That means that an 8-bit timer, counting to 255, can only measure 0.0255ms before it rolls over and starts at zero again. A 16-bit timer, counting to 65535, can measure 6.5535ms. This duration is far too small to use for many real-world applications, so we can increment a counter every time it overflows.

Another useful way to "slow down" the timer is to use a prescale, which effectively divides the internal clock even further. In our examples below we will use a prescale factor of 8 with the 16-bit Timer1. This means that the timer will only increment once every eight ticks, therefore it will overflow after 52.428ms. Here is the formula for calculating a timer's duration:

10^{-7}s * (prescale factor) * (# of ticks) = time elapsed

To calculate the number of ticks to count to achieve a specific duration we solve for (# of ticks):

# of ticks = (time elapsed) * 10^{7}s / (prescale factor)

For our examples we want a nice even interval of 25ms, which we can use to calculate durations of one second. Using the last equation we find that we need 31250 ticks to do this. Since it is easier to look at an overflow condition in the timer, we will preload the timer with a value that will cause this overflow after 31250 ticks. To get this value we subtract 31250 from the 16-bit timer's maximum value (65535) and get 34285. This is the value we will preload our timer with. In the code below you will see 34285 represented in hexadecimal as 0x85ED.

When the timer overflows it will flip a bit, called the interrupt flag. This flag is what we will use in both the polled method and the interrupt-driven method. First we will examine the polled method.

3. Polled Timers: Configuration

We are going to use Timer1 for our examples. Refer to page 135 in the PIC18Fxx20 data sheet for details about this timer.

First we must configure the timer for use. In our example we will perform this at the end of the `User_Initialization` routine in the `user_routines.c` file like this:

```
/* Add your own initialization code here. */
T1CON = 0x30;           /* 1:8 Prescale */  /* see data sheet p.135 */
TMR1H = 0x85;           /* Load high byte of Timer1 */
TMR1L = 0xED;           /* Load low  byte of Timer1 */
T1CONbits.TMR1ON = 1;   /* Turn timer on */
```

There are also C timer functions to perform the same effect. They are documented on pages 47-52 of "MPLAB® C18 C Compiler Libraries". To use these libraries, you must put this line at the top of any .c files which call them:

```
#include <timers.h>
```

Now, instead of manually configuring the timer as in the above lines, you could use these functions:

```
/* Add your own initialization code here. */
OpenTimer1( TIMER_INT_OFF   &
            T1_16BIT_RW     &
            T1_SOURCE_INT   &
            T1_PS_1_8       &
            T1_OSC1EN_OFF   &
            T1_SYNC_EXT_OFF );
WriteTimer1( 0x85ED ) ;        /* Pre-load TMR1 to overflow after 25ms */
```

OpenTimer1 will zero the timer and start it running. That is why you must preload the value after OpenTimer1.

Note that you can turn your timer on or off at will by simply changing the `T1CONbits.TMR1ON` bit to a 1 or 0, respectively. After executing either of these examples, Timer1 will be ticking away, but you could delay its start until later in your program if you wish.

4. Polled Timers: Loop Code

Now that our timer has been started, we will monitor when it overflows. When this happens, the PIR1bits.TMR1IF bit will change from 0 to 1. This is called the Timer1 interrupt flag. Even though we are not using the interrupt-driven method (interrupts have not been enabled), we can still poll the interrupt flag.

To use this timer we will monitor it in the "fast" loop, which is contained in the `user_routines_fast.c` file. First we must declare some variables to use for our counters. Put this at the top of your file:

```
/* DEFINE USER VARIABLES HERE */
unsigned char    t25msDelay;
unsigned char    t100msDelay;
unsigned int     secondCount;
```

Now for the code that does most of the work. Put this in the `Process_Data_From_Local_IO` routine:

```
/* Add code here that you want to be executed every program loop. */
  if (PIR1bits.TMR1IF)        /* Timer1 overflowed */
  {
    PIR1bits.TMR1IF = 0;      /* Clear the interrupt flag */
    TMR1H = 0x85;             /* Reset Timer 1 */
    TMR1L = 0xED;
    if (t25msDelay > 3)
    {
      rc_dig_out03 ^= 1;      /* Pin3 toggles every 100ms */
      t25msDelay = 0;
      if (t100msDelay > 9)
      {
        rc_dig_out04 ^= 1;    /* Pin4 toggles every 1s */
        t100msDelay = 0;
        secondCount++;
        printf("Pin 4 = %d, Elapsed Time (s) =
          %d\n",(int)rc_dig_out04,secondCount);
      }
      t100msDelay++;
    }
    t25msDelay++;
  }
```

Now if you compile and download this project to your Robot Controller you will see a message like this printing out on your terminal screen every second: `Pin 4 = 1, Elapsed Time (s) = 1`

You can also monitor digital outputs 3 and 4 with an oscilloscope and verify that they are toggling every 100ms and 1 second, respectively. If you don't have an oscilloscope, you should still be able to see digital output 4 changing every second on a voltmeter.

5. Using Timers With Interrupts: Configuration

For the interrupt-driven method you must configure your timer in the same manner as for the polled method. Then you must also configure and enable the interrupts. Note the last three lines of code here:

```
/* Add your own initialization code here. */
T1CON = 0x30;           /* 1:8 Prescale */
TMR1H = 0x85;           /* Pre-load TMR1 to overflow after 25ms */
TMR1L = 0xED;
T1CONbits.TMR1ON = 1;   /* Turn timer on */

IPR1bits.TMR1IP = 0;    /* Set Timer1 Interrupt to low priority */
PIE1bits.TMR1IE = 1;    /* Interrupt when Timer1 overflows */
INTCONbits.GIEL = 1;    /* Enable Global Low Priority Interrupts */
```

The last three lines are the only thing added to the configuration that we did for the polled method. These lines simply configure and enable an interrupt condition when Timer1 overflows. In the next section we will see exactly what this means.

6. Using Timers With Interrupts: Interrupt Handler

Timer1 will overflow when its value reaches 0xFFFF. It will automatically roll over to 0 and the Timer1 interrupt flag (`PIR1bits.TMR1IF`) will be set. Since we have enabled the Timer1 interrupt (and global low priority interrupts) in our configuration, whenever `PIR1bits.TMR1IF = 1` an interrupt condition will occur. No matter where the program execution is in your code when this happens, it will immediately drop what it's doing and jump to the `InterruptHandlerLow` routine in `user_routines_fast.c` to handle the interrupt. This is why this method is more accurate and dependable than the polled method.

As in the polled method, we must first declare our variables at the top of our file. In addition to the three counters, we also add a flag called `UpdateDisplay`. For simplicity in our example we are using an entire byte for this value, but ideally you would define a structure elsewhere and make `UpdateDisplay` be only one bit.

```
/* DEFINE USER VARIABLES HERE */
unsigned char    t25msDelay;
unsigned char    t100msDelay;
unsigned char    UpdateDisplay;
unsigned int     secondCount;
```

And now here is the code you should add at the end of `InterruptHandlerLow:`

```
void InterruptHandlerLow ()
{
  unsigned char int_byte;
  if (INTCON3bits.INT2IF)         /* The INT2 pin is RB2/INTERRUPTS 1. */
  {
    INTCON3bits.INT2IF = 0;
  }
...
  else if (PIR1bits.TMR1IF)
  {
    PIR1bits.TMR1IF = 0;          /* Clear the Timer1 Interrupt Flag */
    TMR1H = 0x85;                 /* Reset Timer 1 */
    TMR1L = 0xED;
    if (t25msDelay > 3)
    {                             /* 100ms have passed */
      rc_dig_out03 ^= 1;          /* Pin3 toggles every 100ms */
      t25msDelay = 0;
      if (t100msDelay > 9)
      {                           /* 1 second has passed */
        UpdateDisplay = 1;        /* so we set our own flag */
        t100msDelay = 0;
      }
      t100msDelay++;
    }
    t25msDelay++;
  }
```

While you are inside the interrupt handler every other operation is on hold. Therefore the only important action that we perform is to set the `UpdateDisplay` flag to 1. In the next step we will check this flag outside of the interrupt handler and perform the desired actions.

7. Using Timers With Interrupts: Loop Code

Now, inside the Process_Data_From_Local_IO routine at the end of user_routines_fast.c we will check to see if a second has passed and if so we toggle digital output 4 and print our message to the screen. Notice how we "protect" our `UpdateDisplay` flag by disabling interrupts before we clear it and re-enabling them after. This is another standard practice for using interrupts, so that if another interrupt occurs while we are changing it then it will wait until after we have finished clearing it before we jump to the interrupt handler again.

```
void Process_Data_From_Local_IO(void)
{
  /* Add code here that you want to be executed every program loop. */
  if (UpdateDisplay)
  {
    INTCONbits.GIEL = 0;      /* Disable Low Priority Interrupts */
    UpdateDisplay = 0;
    INTCONbits.GIEL = 1;      /* Enable Low Priority Interrupts */
    rc_dig_out04 ^= 1;        /* Pin4 toggles every 1s */
    secondCount++;
    printf("Pin 4 = %d, Elapsed Time (s) =
      %d\n",(int)rc_dig_out04,secondCount);
   }
}
```

If you compile and run this code, you will see output which is identical to that from the polled method.

8. Conclusion

There are many other ways to use timers and interrupts than what we have shown you here, but hopefully you will see that they are not difficult to use. Using interrupts is the preferred method for achieving flexible, efficient, and powerful code, but when using them you must keep in mind the following points:

First, do not perform extensive operations inside your interrupt handler. You should get in and out of the interrupt handler routine as quickly as possible so that you don't hold up the execution of your main program.

Second, protect variables that are used in the interrupt routines. Anytime a variable which is used in an interrupt routine is modified outside of the interrupt routine (like our `UpdateDisplay` flag) you must protect it by disabling the interrupt, then modifying the variable, and then re-enabling the interrupt. Failing to do this is the most common bug that occurs when using interrupts because you risk corrupting your data and/or causing intermittent anomalous operation.

Finally, whenever you are protecting these variables by turning off interrupts outside of the interrupt handler, you also want to take care of business as quickly as possible, just as you did inside the interrupt handler. Interrupts can be left pending for a short period of time, but they may be dropped if you take too long.

Keeping these points in mind you will easily be able to handle real-time processes and events in code.