Port Triggered Interrupts

What is an Interrupt?

An interrupt is a defined condition that causes the microcontroller (MCU) to suspend what it is doing and go to a known memory location to execute a special block of code. The condition could be that a certain time has elapsed (Timed Interval) or a port (or ports) has changed state (Port Trigger Interrupt). Interrupts are used in computers to ensure that a specific task is done a certain number of times or that an alarm condition is handled IMMEDIATELY.

This chapter covers Port Triggered Interrupts. For a discussion of interrupts that are generated based on a timed event, please see the chapter on RTCC Interrupts.

Overview

Microcontrollers are marvelous little devices that take the drugery out of life. We could either sit at our front door and wait for a thief to break it down or we could connect an MCU to that sensor that will notify us that there is a message for us to handle NOW!!!!

In this chapter I assume you read the previous one on Timed Interrups so that we can get right into Port Triggered Interrupts. Again, we will make a VERY SIMPLE program that only emphasizes the narrow concept being demonstrated. In the case of this chapter we are going to design a program that will respond to a button being pressed and when it is pressed, it will blink an LED and make a sound.

How do I know my program got started?

A couple of years ago I learned a valuable lesson from the masters at Parallax. In several of the demonstration programs, they would instruct the MCU to make a quick sound during the initialization phase so that you know the MCU is working and that your speaker is connected properly. Let's do that first.

In the SX help file under the SOUND command they have a little schematic on how to hook up a piezo element to the MCU. Or if you have a Professional Development Board, just connect port RA.0 to X7 block (lower right) and make sure the volume control is about half way.

```
DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000 ' Identify the frequency of the Resonator

PROGRAM Start
Start:
   tris_A=%0000    ' Set port direction (0=output, 1=input)
   SOUND RA.0, 100, 20 ' Send a tone for 20 * 10 miliseconds to RA.0
Main:
   goto main
```

After you load the program and the MCU starts, it should make a short sound. Ok, now we know that part of the hardware is connected correctly.

Now let's test the button!

One of the nice things about using the Professional Development Board (PDB) is that several of the peripherals are already wired up for you. Referring to the schematic for the PDB on pg 4, the buttons are connected to Vss (ground) on one side and "pulled high" on the other. This means that the other side of the push button is connected to a resistor which is then connected to Vdd (5vdc). This means that if connected to an MCU that it will show a logical ZERO when you press the button and ONE when you let go.

Let's connect a push button to our MCU pin RB.0. If you have a PDB, just run a wire from RB.0 to the X10 block, position 0. This enables the "0" pushbutton on the board. If you do not have a professional development board, you can use any push button and connect it as follows:

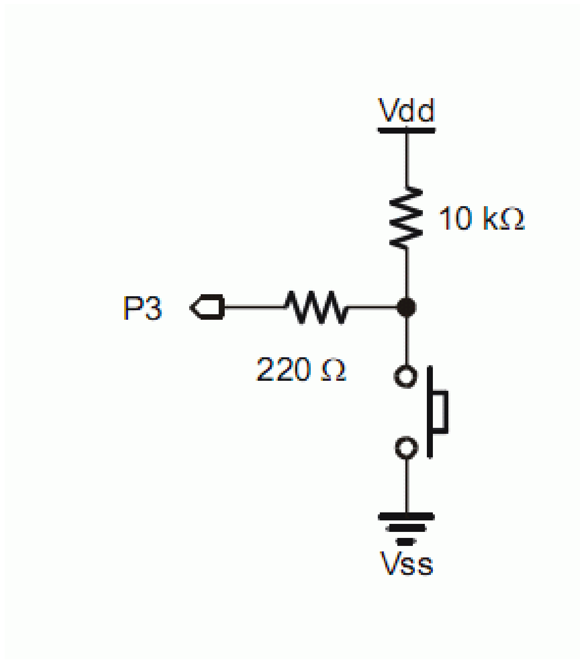

Diagram from "What's a Microcontroller?, pg 69 (c) 2009 Parallax Corp

Port B is used for Hardware Interrupts

On the SX processor, the RB port is special in that it is the one used for port triggers. Thus, the selection of RB.0 is not an arbritrary one.

For the following program, we will just use simple logic to test the condition of the port. In other words, when you press the button, the main loop will test the condition of the button, if it is showing that the button is connected to ground (someone is pushing it), it will sound the tone, otherwise it will ignore it and loop again.

To make this program work properly we need to indicate to the compiler that we want to use the RB.0

pin as an INPUT. We do that with the "tris" compiler directive. This command indicates the "direction" of a specific pin. One indicates input and zero indicates output. If you imagine that the one looks like an "I" for input and the zero looks like the "O" for output, they you have it memorized already.

```
    tris_B=%00000001  'Set port direction (0=output, 1=input)
```

Next we will add code that tests the condition of RB.0. Is it a one or a zero? Remember zero indicates that the button has been pushed and is connected to ground.

```
    DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
    FREQ            4_000_000 ' Identify the frequency of the Resonator

    Program Start
    Start:
       ' initialization code here
       tris_A=%0000        ' Set port direction (0=output, 1=input)
       tris_B=%00000001    ' Set port direction (0=output, 1=input)
       SOUND RA.0, 100, 20 ' Send a tone for 20 * 10 miliseconds to RA.0

    Main:
       ' Later we will put additional code here
       ' For now we just want to loop forever
       IF RB.0 = 0 THEN
          sound RA.0,75,50
       ENDIF
       goto main
```

When you start the MCU, you will get one tone. When you press the button you will get a different tone. This is because we changed the parameters of the SOUND command. Here it says RA.0,75,50 which means send the sound to the RA.0 pin, using the 75 tone (which is lower than the 100 tone) and for 50*10 milliseconds.

Interrupt Etiquette

Ok, we know our basic hardware works. Now all we need to do is add the interrupt code. What we want to be able to do is to take out the trigger code in the main loop and put it into the interrupt. A couple of things to remember though:

- Whatever you put into the interrupt section, it should execute quickly. This will become VERY important if your main loop is doing something that is time sensitive like monitoring a serial port.

- If you want to monitor more than one "alarm" line, you will need to set a variable in the interrupt so that the main loop knows that an additional alarm was triggered.

- Just like your alarm clock, once an alarm is triggered, you have to remember to turn it off. I'll demonstrate that below.

Revisit the OPTION Register

Option Register							
7	6	5	4	3	2	1	0
RTW	**RTI**	RTS	**RTE**	PSA	PS2	PS1	PS0

Here we are going to set the OPTION register a little differently. Basically we want to tell the compiler that do not need timed interrupts and therefore we do not need to use the prescaler:

```
7 RTW - 1 = use the RTCC (not the watchdog timer)
6 RTI -  1 = disable interrupt on rollover
5 RTS -   0 = increment on instruction cycle
4 RTE -    1 = we want the HIGH to LOW transition (default)
3 PSA -     0 = turn the prescaler off.  We want to know NOW, not later!
2 PS2 -      0 = we are not using the prescaler.
1 PS1 -       0 = we are not using the prescaler.
0 PS0 -        0 = we are not using the prescaler.
```

This all is accomplished by using the OPTION command:

```
OPTION = %11010000 'RTCC,no rollover,Inst Cycle, High to Low
' or you could use the HEX version
OPTION = $D0
```

Now Let's Add Some Interrupt Code

As we discussed in the last chapter, most of the action happens after the FREQ command and before the START. In the code snippet below is an outline of what we are going to do. We still need to fill in the INTERRUPT and MAIN areas with code but it gives you an idea of what it looks like.

```
DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000 ' Identify the frequency of the Resonator
INTERRUPT
   ISR_Start:
      ' Do this when an alarm is triggered
   ISR_Exit:
      ' Before you exit the interrupt, reset the
      ' alarm or it will just come right back.
      ' This is known as the "pending" register.
      WKPND_B = 0  ' reset condition that caused the interrupt
   RETURNINT ' {cycles}

PROGRAM Start
   Start:
      ' Setup Port A (our speaker)
      tris_A = %0000      ' Set port direction (0=output,1=input)

      ' Setup Port B (our sensors)
      tris_B = %00000001  ' Set pin RB.0 as an input
      wken_B = %11111110  ' Set pin RB.0 as being able to trigger
                          ' an interrupt.

      ' Setup Port C (my LED outputs)
      tris_C = %00000000  ' Set all of RC as an output

      ' Set the Option register
      OPTION = %11010000  ' (see below)
                          ' Use RTCC and DISABLE rollover interrupt

      SOUND RA.0, 100, 20 ' Send a tone for 20 * 10 miliseconds to RA.0

Main:
  ' Later we will put additional code here
  ' For now we just want to loop forever
  goto Main
```

Port Triggered Interrupt Code

We want to be able to detect when someone pushes a button. When they do, we want to jump to the interrupt code and do something that will get our attention. We want to be careful that what we do in the interrupt so that it does not take up too much time and impact what we are doing in the main loop.

```
DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000 ' Identify the frequency of the Resonator
INTERRUPT
   ISR_Start:
      ' Do this when an alarm is triggered
      RC.0 = ~ RC.0       ' Blink the LED by reversing its state
      SOUND RA.0, 75,5    ' Send a tone for 5 * 10 miliseconds to RA.0 (short interval)
   ISR_Exit:
      ' Before you exit the interrupt, reset the
      ' alarm or it will just come right back.
      ' This is known as the "pending" register.
      WKPND_B = 0  ' reset condition that caused the interrupt
   RETURNINT ' {cycles}

PROGRAM Start
   Start:
      ' Setup Port A (our speaker)
      tris_A = %0000      ' Set port direction (0=output,1=input)

      ' Setup Port B (our sensors)
      tris_B = %00000001  ' Set pin RB.0 as an input
      wken_B = %11111110  ' Set pin RB.0 as being able to trigger
                          ' an interrupt.

      ' Setup Port C (my LED outputs)
      tris_C = %00000000  ' Set all of RC as an output

      ' Set the Option register
      OPTION = %11010000  ' (see below)
                          ' Use RTCC and DISABLE rollover interrupt

      SOUND RA.0, 100, 20 ' Send a tone for 20 * 10 miliseconds to RA.0

Main:
  ' Later we will put additional code here
  ' For now we just want to loop forever
  goto Main
```

Debouncing the Button

There are a couple of problems with this code in real life. First of all, the button will "bounce". As you were testing it, you noticed that the LED sometimes flashes quickly on and then off. Other times the LED smoothly transitioned from on to off with each push of the button. This is known a "switch bounce" and happens because the MCU is SO FAST that it actually detected the brief interval when the button was just barely touching and a tiny bit of current got through. Then it detected when you actually held the button down. Thus, it detected the equivalent of TWO button pushes. We can solve the problem with hardware by adding some capicators and delay circuts or we can solve it with sofware by telling the MCU that we will only respond to an alarm once every so many milliseconds. For now, and for simplicity, we will just use two pushbuttons.

Hardware Connections

Connect the following items to your microcontroller:

- RA.0 is connected to the Piezo speaker
- RB.0 is connected to a push button that is grounded when pushed and "tied-high" when released.
- RB.1 is connected to a second push button that is also grounded and tied high.
- RC.0 is connected to a resistor (about 220 ohms) which is then connected to an LED. The other side of the LED is grounded.

Taking too long in the Interrupt

The SOUND RA.0, 75,5 basically "hogs" the MCU for a period of time. This is bad in an interrupt because you want the MCU to be handle several things and not miss a beat. A better solution would be to put the SOUND command in the MAIN loop and have the interrupt simply set a variable in the interrupt to indicate that an alarm condition exists. This will enable the interrupt to get in and out quickly and the main routine to handle the alarm. We'll also solve the switch "bounce" by using two push buttons, one to indicate it is on, and one to indicate it is off. Not the best solution, but the main emphasis for this tutorial is simplicity.

Going Further

Another use for a port trigger is to develop an accurate clock. Many GPS boards output a super accurate pulse on the second. You can use that pulse to trigger an interrupt that will reset an internal timer once a second.

Well now that you know how to trigger an interrupt using an outside trigger tied to a port you can create all sorts of fun projects.

Revised Port Triggered Interrupt Code

```
DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000 ' Identify the frequency of the Resonator

Alarm   VAR Byte

INTERRUPT
   ISR_Start:
      ' Do this when an alarm is triggered
      if RB.0 = 0 THEN
         Alarm = 1
      ENDIF
      if RB.1 = 0 THEN
         Alarm = 0
      ENDIF
   ISR_Exit:
      ' Before you exit the interrupt, reset the
      ' alarm or it will just come right back.
      ' This is known as the "pending" register.
      WKPND_B = 0  ' reset condition that caused the interrupt
   RETURNINT ' {cycles}

PROGRAM Start
Start:
   ' Setup Port A (our speaker)
   tris_A = %0000      ' Set port direction (0=output,1=input)

   ' Setup Port B (our sensors)
   tris_B = %00000011  ' Set pin RB.0 as an input
   wken_B = %11111100  ' Set pin RB.0 as being able to trigger
                          ' an interrupt.

   ' Setup Port C (my LED outputs)
   tris_C = %00000000  ' Set all of RC as an output

   ' Set the Option register
   OPTION = %11010000  ' (see below)
                       ' Use RTCC and DISABLE rollover interrupt

   SOUND RA.0, 100, 20 ' Send a tone for 20 * 10 miliseconds to RA.0
   Alarm = 0

Main:

   IF Alarm = 1 then
      RC.0 = 1         ' Turn the ALARM LED ON
      SOUND RA.0, 75, 10  ' Make a sound
   ELSE
      RC.0 = 0
   ENDIF

  ' Later we will put additional code here
  ' For now we just want to loop forever
  goto Main
```

*File: **./sx/Interrupts-Port.mkd** updated: 04/23/2010 17:30*