Chapter 11: Interrupt On Change

The last two chapters included examples that used the external interrupt on Port C, pin 1 to determine when a button had been pressed. This approach works very well on most PIC microcontrollers, but with the 16F1459 a problem arose in that we were unable to use the in circuit debugger when using this approach because the debugger also uses the external interrupt pin.

There is another interrupt, however, that allows you to detect changes on any pin on Port A or Port B that can be used to accomplish the same purpose. As will be seen towards the end of this chapter, this pin can also be used to read keyed input from a standard keypad.

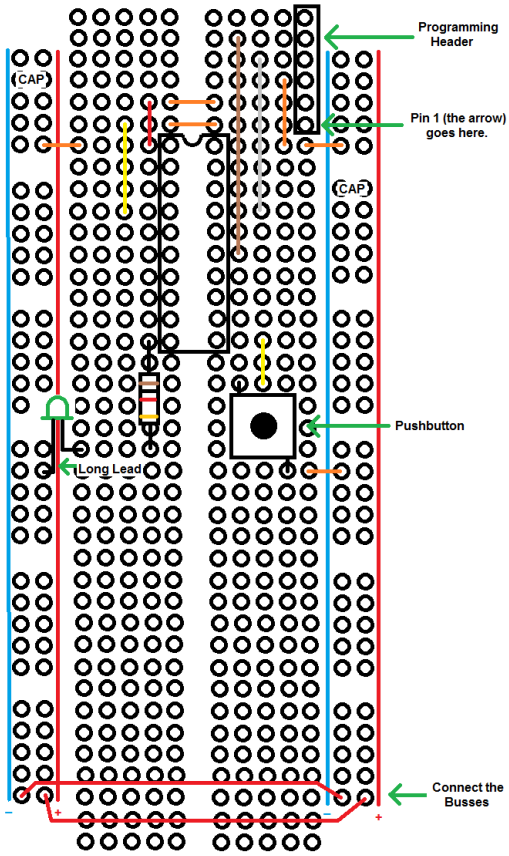

Figure 1: Board Set Up for Interrupt on Change on Pin RB6.

Start by redoing the breadboard so that the button is connected to Port B, pin 6. Any pin on Port A or B could be used, but this one is the same pin that we used in chapter 7 and results in only a minor change in the board setup from chapter 10. Note that the pushbutton is connected to ground, not the positive rail. This will allow us to once again use the internal weak pull-up resistor in the PIC rather than adding one of our own. Recall that enabling the weak pull-up resistor involves clearing pin 7 of the option register and then turning on the pull up for pin 6 of Port B by setting pin 6 of the WPUB register to 1.

As in the case of the other interrupts that we have used, two bits in the INTCON register must be enabled to turn on the interrupt. In this case, in addition to the GIE bit (bit 7), we must also turn on the IOCIE bit (bit 3). We can do this either by setting the two individual bits or writing the binary value **0b10001000** to the INTCON register. This simply turns on the interrupt on change in general, it does not cause any specific pin to generate an interrupt.

We can cause an interrupt to occur either when the pin transitions from low to high (rising edge) or from high to low (falling edge) or both. To detect a rising edge, we set the pin (pin 6) in the IOCBP register. To detect a falling edge, we set pin 6 in the IOCBN register. If you are programming in assembly language, it will be important to note that these registers are both in bank 7, so you will need to change to that bank before setting the pins. If you are programming in C, this will be handled automatically by the compiler. Let's have the interrupt fire when the button is released. At that point the pin will be pulled back high by the pull-up resistor so the transition will be from low to high. So IOCBP should be set to **0b01000000**.

Here is the revised initialization code to trigger an interrupt when the button is pressed using the interrupt on change interrupt:

```
TRISB  = 0b01111111;          //output on RB7, input on the others
OSCCON = 0b00110100;          //4 MHz clock
ANSELB = 0;                   //all digital pins
OPTION_REG = 0b00000000;  //bit 7 clear to turn on pull-ups
WPUB = 0b01000000;            //RB6 pull-up turned on.
GIE = 1;                      //global interrupts enabled
IOCIE = 1;                    //interrupt on change enabled
IOCBP = 0b01000000;           //pin RB6 triggers the interrupt
```

When the falling edge occurs signaling the button has been pressed, the requisite pin in the IOCBF register (or IOCAF, if Port A) will be set. Assuming the interrupt is turned on in the INTCON register, program execution will immediately go to the interrupt service routine. As with all interrupts, it will be necessary to clear the flag in the interrupt service routine. You could do this simply by writing 0x00 to IOCBF, but if another interrupt on change occurs on another Port B pin while you are in the interrupt service routine, you'll clear that interrupt as well and will never find out that it occurred. It is not uncommon to use multiple interrupt on change interrupts in the same program. As a result, it is a good idea to first determine which pin of IOCBF caused the interrupt and to clear that pin.

In assembly language, this could be done as:

```
movlw   0xff
movlb   7
xorwf   IOCBF, W
andwf   IOCBF, F
movlb   0
```

The first line puts all ones in the working register. The second line switches to bank 7 where the IOCBF register is located. The third line takes the IOCBF register and does an exclusive or with that and the working register. Suppose the pin 6 flag of IOCBF was a one. This operation would then work as follows:

```
Working register:   1111 1111
IOCBF register:     0100 0000
xorwf result:       1011 1111
```

Because of the 'w' after the comma in the xorwf instruction, the result is placed in the working register. The next line does a bitwise AND operation on the value in the working register and the value in IOCBF:

```
Working register:   1011 1111
IOCBF register:     0100 0000
AND result:         0000 0000
```

The register is cleared. But what if another bit gets set while the operation is going on because another interrupt on change operation occurs? This last example would change to:

```
Working register:   1011 1111
IOCBF register:     0100 0100   note that bit 2 has just changed
AND result:         0000 0100
```

The new flag would be preserved. In C this could be written:

```
char x;
x = IOCBF ^ 0xFF;
IOCBF = x & IOCBF;
```

Note that we need a variable (x) to hold the value of the exclusive or calculation so that it can be used in the next line of code.

The complete interrupt service routine would be:

```
void interrupt isr(){
        char x;
        RB7 = RB7 ^ 1;      // flip the state of the button.
        __delay_ms(10);     // debounce the button.
        x = IOCBF ^ 0xFF;
        IOCBF = x & IOCBF;  / /clear the interrupt flag
}
```

The complete program in C would be:

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>

//pragma statements to set configuration bits go here…

#define _XTAL_FREQ 4000000

void initial();
void interrupt ISR(void);

int main(int argc, char** argv) {
  initial();
  while (1);
  return (EXIT_SUCCESS);
}
void initial(){
   TRISB  = 0b01111111;           //output on RB7, input on the others
   OSCCON = 0b00110100;           //4 MHz clock
   ANSELB = 0;                    //all digital pins
   OPTION_REG = 0b00000111;  //bit 7 clear to turn on pull-ups; 1:256 prescaler
   WPUB = 0b01000000;             //RB6 pull-up turned on.
   GIE = 1;                       //global interrupts enabled
   IOCIE = 1;                     //interrupt on change enabled
   IOCBP = 0b01000000;            //pin RB6 triggers the interrupt
}

void interrupt isr(){
   if (IOCBF6){
      char x;
      RB7 = RB7 ^ 1;      // flip the state of the LED.
       __delay_ms(10);
      x = IOCBF ^ 0xFF;
      IOCBF = x & IOCBF;  //clear the interrupt flag

   }
}
```

Here is the code to accomplish the same thing in assembly language. The values in the delay loop were determined experimentally, using the simulator and stopwatch to produce a delay of 10 ms.

```
      radix hex
      include "p16f1459.inc"
      errorlevel -302
      __CONFIG _CONFIG1, _FOSC_INTOSC & _WDTE_OFF & _PWRTE_OFF & _MCLRE_ON & _CP_OFF &
            _BOREN_ON & _CLKOUTEN_OFF & _IESO_OFF & _FCMEN_OFF
      __CONFIG _CONFIG2, _WRT_OFF & _CPUDIV_CLKDIV6 & _USBLSCLK_48MHz & _PLLMULT_3x &
            _PLLEN_DISABLED & _STVREN_ON & _BORV_LO & _LPBOR_OFF & _LVP_OFF

    org 0
    goto MAIN
    org 4
    goto ISR
count equ 20

MAIN
   call INITIAL
   goto MAIN

INITIAL
   movlb 1                          ;switch to bank 1
   movlw b'01111111'
   movwf TRISB                      ; bits 0 through 6 are all set to inputs.  Bit 7 is an output.
   movlw b'00110100'
   movwf OSCCON                     ; set internal oscillator speed.
   bcf OPTION_REG, NOT_WPUEN        ; enable weak pull-ups.  Note this is the bit name from the p16f1459.inc file
   movlb 3                          ; switch to bank 3
   clrf ANSELB                      ;all digital pins on Port B
   movlb 4
   bsf WPUB, WPUB6                  ;RB6 pull-up turned on
   bsf INTCON, IOCIE                ;turn on interrupt on change
   bsf INTCON, GIE                  ;turn on global interrupts
   movlb  7                         ;switch to bank 7
   movlw  0b01000000
   movwf  IOCBP                     ;turn on interrupt on change for RB6 rising edge
   movlb 0                          ; move back to Bank 0.
   return

ISR
   movlw  0b01000000
   xorwf    PORTB, f                ; flip the state of pin RB6
   movlw  0xff
   xorwf  IOCBF, W
   andwf  IOCBF, F                  ;clear the interrupt flag
   retfie

DEBOUNCE
   movlw 0xFF
   movwf count
   movlw 0x0D
   movwf count2
DELAYLOOP
   decfsz count, f
   goto DELAYLOOP
   decfsz count2, f
   goto DELAYLOOP
   return


   end
```

Debouncing the Button Using an Interrupt

It's not a good idea to put a delay loop in an interrupt service routine. The whole point of using an interrupt is to free up processor cycles to do other things rather than constantly checking to see if a button has been pressed or a timer has expired. It might look to you as if the debounce routine is not part of the interrupt service routine because it appears to be separate. But since it is

being called from the isr, it is part of the isr. This is fairly obvious when you look at the C code rather than the assembly code.

The debounce routine in the above code isn't horrible because it uses a pretty short delay, but it is possible to eliminate it by having the delay handled by a timer interrupt. The strategy here is to turn off the button interrupt in the code that services the button press and turn on the timer0 interrupt. When the timer0 interrupt expires, turn the button interrupt back on and turn the timer interrupt off. This way any button actions that may occur before the timer rolls over will be ignored.

If we set the timer0 prescaler for 1:256, the timer will roll over approximately once every 195 ms. That will allow plenty of time for the button to stop bouncing. It would take significant coordination to press the button twice in 195 ms, so it is rather unlikely that any user input will be lost setting the timer for this length of time, though if that did turn out to be a problem, we could just change the prescaler to 1:128 to fix it.

Only one change needs to be made to the initialization code from the previous example; the option register must be configured to set the prescaler:

```
OPTION_REG = 0b00000111;
```

Here is the revised interrupt service routine:

```
void interrupt isr(){
   if (IOCBF6){
      char x;
      RB7 = RB7 ^ 1;          // flip the state of the LED.
      x = IOCBF ^ 0xFF;
      IOCBF = x & IOCBF;      //clear the interrupt flag
      IOCIE = 0;              //temporarily disable interrupt on change
      TMR0 = 1;               //reset TMR0 so it will take a while to trigger
      TMR0IE = 1;             //enable the TMR0 interrupt
   }
   else if (TMR0IF){
      IOCIE = 1;              //re-enable interrupt on change
   }
   TMR0IF = 0;                //always clear the timer flag
}
```

The first thing to notice is that there is no longer a delay function call. This means that execution will get in and out of the isr extremely quickly. Other than that, the first four lines of code are the same as in the previous version. They flip the state of the LED and turn off the change on interrupt flag.

Then we disable the interrupt on change and enable the timer0 interrupt. There is one other line of code here that requires some explanation. Timer0 is constantly counting up. Because the button interrupt could have occurred at any time, we have no idea what the value of the Timer0 register is when the isr is executing. It is possible that it is about ready to roll over. However we want it to count all the way up (delaying the full 195 ms) before turning back on the interrupt on change interrupt. So we set the value of that register to 1 so that it will have to count from 1 all the way up to 255 before rolling over to 0.

It is important to understand the difference between interrupt flags and interrupts. Whenever timer0 rolls over from 255 to 0 the interrupt flag (TMR0IF) gets set. This happens **even if the timer0 interrupt has not been enabled.** The interrupt enable bit only controls whether program execution jumps to the interrupt service routine, not whether the flag gets set. This is true of all interrupts, not just the timer0 interrupt. As a result, it is important to use the *if…else if* construction in the isr, not just a series of if statements. The reason for this is that because timer0 rolls over every 195 ms, it is almost certain that when the button is pressed the TMR0IF will be a 1. Since the timer interrupt enable bit is not set, it will not trigger the interrupt though. However, the button will send execution to the isr and once there, if the program checks to see if the timer interrupt flag is set, it will find that it is set. So it is important, if the interrupt was triggered by a button press, to NOT check to see if the timer flag is set. Hence the *if... else if* construction.

Also, it is critical to turn off the timer0 flag before leaving the isr, even if the isr did not handle a timer0 interrupt. The reason for this is the button interrupt handler turns on the timer0 interrupt. If we exit the isr without turning off the timer0 flag, it will trigger an interrupt immediately, because the flag will certainly be set by then.

This may all seem somewhat confusing. If you find it to be so, carefully re-read the last few paragraphs. Remember, the timer0 flag will be set whenever timer0 rolls over from 255 to 0, **regardless of whether the interrupt has been turned on or not.**

Using an Interrupt on Change Interrupt to Read a Keypad

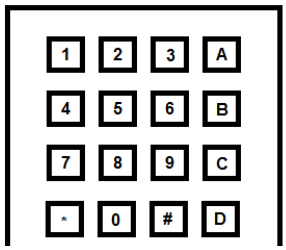

Figure 2: Standard Hex Keypad

Figure 2 shows the layout of a standard keypad that has 16 keys. It looks a lot like a telephone keypad, but has an extra column of keys. These keypads are relatively easy to find[1] and provide an easy and inexpensive way to provide 16 buttons instead of just one. Because there are 16 buttons, one easy way to design this would be to have 32 pins (two for each switch). However, because all the switches would probably have one side connected to ground, we could have done it with just 17 pins, one for each switch and a 17th connected to the ground side on all of the switches.

Reading such a keypad would require a rather large number of microcontroller pins, however, so most of the time keypads are not designed this way. Instead, they are designed as shown in Figure 3. Here, all of the buttons in each column are connected together (the red lines) and all of the buttons in each of the rows are connected together. Then each of these 8 lines would be connect to a pin on the PIC. Note that each button on the keypad is connected to a unique combination of two pins. No button is connected to the same two pins as any other button. If we sequentially apply a positive voltage to pin 1, then pin 2, then pin 3 then pin 4 and look for a voltage on pins 5, 6, 7, and 8, we can determine which key (if any) has been pressed. If no key is pressed applying a voltage to the first four pins will not result in any voltage appearing on any of the second four pins. But suppose the '5' key is pressed. When a voltage is applied to pin one,

[1] See for example: http://www.bgmicro.com/SWT1067.aspx

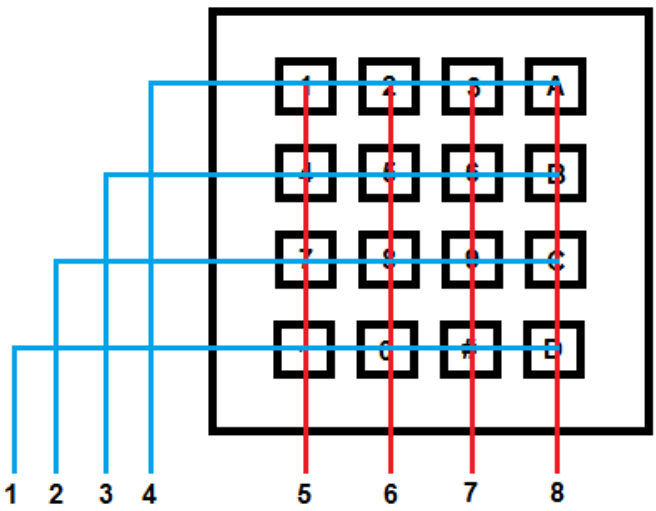

Figure 3: Matrix Keypad Connections

no voltage will be detected on any of the second four pins. The same thing will be the case when a voltage is applied to pin 2 or pin 4. But when a voltage is applied to pin 3, we'll detect a voltage on pin 6 and only pin 6 and that will tell us that the '5' key has been pressed.

In practice, in order to use the internal pull ups, we'll setup the PIC pins connected to lines 1–4 as outputs (do this with four pins on Port C) and the PIC pins connected to lines 5–8 will be setup as inputs (do this with four pins on Port B). We'll use Port B for the inputs because we will want to use the weak pull-ups that are available on that Port.

The only pins available on Port B are RB4, RB5, RB6, and RB7. Because we'll be using the internal pull-ups, we'll actually want to set all of the output pins (1-4) to high and then sequentially flip them one by one to low. Each time we move one of the pins to low, we'll look at input pins 5-8 and see which pin, if any, is low to determine which key has been pressed.

This process could be implemented through polling (repeatedly checking to see what key had been pressed). Alternatively, this situation is an excellent opportunity to use the interrupt on change interrupt. Turn on the interrupt on change interrupt for pins RB4, RB5, RB6, RB7. We will want that interrupt to trigger when the key is pressed, not when it is released because while it is still being pressed, we will have to determine which key has been pressed.

Then set the pins connected to lines 1-4 **all low**. Whenever a key is pressed, there will be a change from high to low on one of the four input pins. The interrupt service routine will then have to step through the pins connected to lines 1-4 sequentially making each low in order to determine which key has been pressed. It will have to accomplish this while the key is still being pressed or the information will be lost. But this can be done in a few microseconds so that the users finger will not have a chance to let go of the key before the information is recorded.

TRY THIS: Connect a keypad to your PIC as described above. Move the LED to a pin that is not used by the keypad. Have the PIC read the key that is pressed and flash the LED the number of times represented by the key. For the non-numeric keys, use the following:

A = 10
B = 11
C = 12
D = 13
* = 14
# = 15