Chapter 10

Interrupts

If we are carrying out a task and we are interrupted by the phone ringing. We answer the phone, deal with it, and then return to our task. We need to remember of course what position we were at in the task. We could have been interrupted by a knock on the door.

The microcontroller does a similar thing when handling an interrupt. The microcontroller stops the task it is on and deals with the interrupt routine. It remembers where it was and returns back to the task.

There are several ways a microcontroller like the 18F1220 can be interrupted and they are:

- RB0 going high or RB0 going low.
- RB1 going high or RB1 going low.
- RB2 going high or RB2 going low.
- RB4, RB5, RB6, or RB7 changing.
- An analogue measurement being made.
- TMR0 overflowing.
- EUSART receiving.
- EUSART transmitting.
- Capture Compare
- TMR1 overflowing
- TMR2 output matches PR2 (period register).
- TMR3 overflowing.
- Oscillator failing.
- Data EEPROM write complete.
- Low Voltage Detection.

Interrupts can have a high priority or a low priority. High priority interrupts can interrupt low priority interrupt routines. The default condition is that interrupts do not have a priority and this is what we are using here, so there is no requirement to set priority.

When using interrupts they are turned on by their individual IE, interrupt enable bit. All interrupts that are enabled can be disabled and enabled again by the GIE, global interrupt enable bit.

Interrupts have their individual IF, interrupt flag, which is set when that particular interrupt is responsible for the interrupt. Checking the IF will determine which interrupt interrupted the program.

When an interrupt is triggered the corresponding IF is set, the GIE disables all interrupts stopping any further interruptions (no priority is given here). The program breaks to the Interrupt Service Routine (ISR) and executes it from memory location 0×08. The ISR must be loaded into memory starting at location 0×08. At the end of the ISR the IF, interrupt flag, must be cleared in software to stop that flag causing further interrupts. The GIE is reset automatically ready for further interruptions.

The bits used to control the interrupt process for all interrupts can be found in registers:

- RCON
- INTCON
- INTCON2
- INTCON3
- PIR1, PIR2
- PIE1, PIE2
- IPR1, IPR2

In order to understand the application of interrupts we will look at two examples:

Using RB4-7 change and TMR0 overflowing.

RB4-7 change.

We will consider a program being interrupted when RB7 changes state. The program is turning on a single output in turn on RB0—RB6 at 2 s intervals. When interrupted by RB7 changing then RB0—RB6 all flash on and off twice for 1 s. The program then reverts to its task exactly where it was when it was interrupted.

The circuit for this is shown in Figure 10.1.

The program for this system, **RBinterrupt.C,** is shown below.

```
1.  //RBinterrupt.C 7-2-12 DW Smith.
2.  #include <p18f1220.h>
3.  #pragma config WDT=OFF, OSC=INTIO2, PWRT=ON, LVP=OFF, MCLRE=OFF
4.  #include <delays.h>
5.  #pragma interrupt isr            // declares isr as my interrupt (service) routine.
6.  int PORTBtemp;                   //the state of PORTB is stored here on interrupt.
7.  #pragma code isr = 0×08          //puts isr in memory location 0×08
8.  void isr(void)
9.  {
10.     PORTBtemp=PORTB;
11.     PORTB=0xFF;
12.     Delay100TCYx(78);
13.     PORTB=0;
```

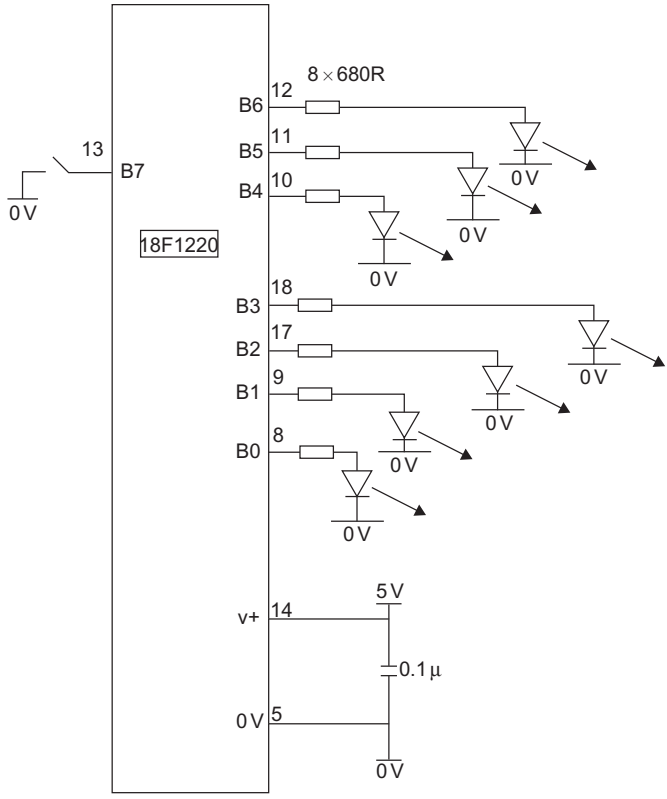

FIGURE 10.1 The PORTB,7 interrupt change circuit.

```
14.         Delay100TCYx(78);
15.         PORTB=0xFF;
16.         Delay100TCYx(78);
17.         PORTB=0;
18.         Delay100TCYx(78);
19.         PORTB=PORTBtemp;
20.         INTCONbits.RBIF=0;              // clear RBIF
21.   }
22.   #pragma code                          //ends code section
23.   void main (void)
24.   {
25.         //SET UP
26.         // OSCCON defaults to 31 kHz. So no need to alter it.
27.         ADCON1=0x7F;                    //all IO are digital or 0b01111111 in binary
28.         TRISA=0b11111111;               //sets PORTA as all inputs
29.         PORTA=0b00000000;               //turns off PORTA outputs, not required, no outputs
30.         TRISB=0b10000000;               //sets PORTB,7 is I/P
31.         PORTB=0b00000000;               //turns off PORTB outputs, good start position
32.         INTCON2bits.RBPU=0;
33.         INTCON=0b10001000;              //sets GIE, RBIE
```

```
34.   while (1)
                {
35.             PORTB=0b00000001;
36.             Delay100TCYx(156);
37.             PORTB=0b00000010;
38.             Delay100TCYx(156);
39.             PORTB=0b00000100;
40.             Delay100TCYx(156);
41.             PORTB=0b00001000;
42.             Delay100TCYx(156);
43.             PORTB=0b00010000;
44.             Delay100TCYx(156);
45.             PORTB=0b00100000;
46.             Delay100TCYx(156);
47.             PORTB=0b01000000;
48.             Delay100TCYx(156);
49.             }
50.   }
```

When the interrupt runs the program needs to remember where it was when the interrupt occurred

EXPLANATION OF THE PROGRAM RBINTERRUPT.C

- Lines 1–4 have been used and discussed previously.
- Line 5. The command **#pragma interrupt isr** declares that **isr** is an interrupt routine. This instructs the micro to save some of the internal registers so that it can carry on from where it left off and set the GIE when returning from the interrupt.
- Line 6 and 10. The interrupt routine is changing PORTB. So we must remember what was happening to PORTB so that we can return and carry on as before. So we store PORTB in PORTBtemp.
- Line 7 writes the ISR into memory location 0×08 where the program starts executing from.
- Line 22 ends the code section placement.
- Lines 8–21 executes the ISR.
- Line 19 restores the saved PORTB conditions back into PORTB.
- Line 20 resets the RBinterrupt flag ready for more interrupts here.
- Line 21 is the end of the ISR and we return back to our main program. The GIE is then set enabling all the interrupts that have been switched on.
- Lines 25–33 is our setup procedure. Note line 30 turns the pull ups on so we don't need a pull up resistor on RB7.
- Line 33 sets the GIE and RBIE bits to enable the interrupt.
- Lines 34–49 is the main light sequencing routine.

The interrupt bits used in this program, GIE, RBIE, and RBIF, can be found in bits 7, 3, and 0 of the interrupt control register, INTCON, as shown in Figure 10.2.

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/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

FIGURE 10.2 INTCON register.

TMR0 OVERFLOWING INTERRUPT

In Chapter 9 we saw how to write a 1 min delay routine. Suppose we want to interrupt a program every minute. We will use the sequencing routine and circuit used in the RBchange interrupt routine.

The code for this circuit is shown below in **TMR0interrupt.C**

//TMR0interrupt.C

```
1.  #include <p18f1220.h>
2.  #pragma config WDT=OFF, OSC=INTIO2, PWRT=ON, LVP=OFF, MCLRE=OFF
3.  #include <delays.h>
4.  #pragma interrupt isr              // declares isr as my interrupt (service) routine.
5.  int PORTBtemp;                    //the state of PORTB is stored here on interrupt.
6.
7.  #pragma code isr=0×08              //puts isr in memory location 0×08
8.  void isr(void)
9.  {
10. PORTBtemp=PORTB;
11. PORTB=0xFF;
12. Delay100TCYx(78);
13. PORTB=0;
14. Delay100TCYx(78);
15. PORTB=0xFF;
16. Delay100TCYx(78);
17. PORTB=0;
18. Delay100TCYx(78);
19. PORTB=PORTBtemp;
20. INTCONbits.TMR0IF=0;              // clear TMR0IF
21. TMR0H =0xF8;
22. TMR0L =0x7F;
23. }
24. #pragma code//ends code section
25. void main (void)
26. {
27. //SET UP
28. ADCON1=0x7F;                      //all IO are digital or 0b01111111 in binary
29. TRISA=0b11111111;                 //sets PORTA as all inputs
30. PORTA=0b00000000;                 //turns off PORTA outputs, not required, no outputs
31. TRISB=0b10000000;                 //sets PORTB,7 is I/P
32. PORTB=0b00000000;                 //turns off PORTB outputs, good start position
33. T0CON=0b10000111;   //sets TMR0 on, 16bits, internal clock, prescaler assigned, prescaler=256
34. INTCON=0b10100000;                //sets GIE,TMR0IE
35. INTCONbits.TMR0IF=0;              // clear TMR0IF
36. TMR0H =0xF8;
37. TMR0L =0x7F;
```

```
38.   while (1)
39.       {
40.       PORTB=0b00000001;          // all outputs off
41.       Delay100TCYx(156);
42.       PORTB=0b00000010;          // all outputs off
43        Delay100TCYx(156);
44        PORTB=0b00000100;          // all outputs off
45.       Delay100TCYx(156);
46.       PORTB=0b00001000;          // all outputs off
47.       Delay100TCYx(156);
48.       PORTB=0b00010000;          // all outputs off
49.       Delay100TCYx(156);
50.       PORTB=0b00100000;          // all outputs off
51.       Delay100TCYx(156);
52.       PORTB=0b01000000;          // all outputs off
53.       Delay100TCYx(156);
54.       }
55.   }
```

Explanation of the Code

TMR0interrupt.C is similar to **RBinterrupt.C** with the following exceptions:

We are of course using the TMR0 interrupt bits instead of the RBchange interrupt bits.

- Line 20 clears the TMR0 interrupt flag, ready for the next 1 min interrupt.
- Lines 21 and 22 set TMR0H to 0xF8 and TMR0L to 0x7F when TMR0L is written to. The 16 bit TMR0 overflows at 0xFFFF setting it initially to 0xF87F means it will run for 0×0780 ($0 \times 0780 = 1920\text{d} = 32 \times 60 = 60\,\text{s}$) before overflowing and interrupting.
- Line 35 clears the TMR0 interrupt flag.
- Lines 36 and 37 set TMR0H to 0xF8 and TMR0L to 0x7F, to preload TMR0 before starting the main program, lines 38–54.

A 10 min Interrupt

To set TMR0 to overflow after say 10 min (600 s).

At 32 timing pulses per second we would have $600 \times 32 = 19{,}200$ pulses.

19,200 in hex is $0 \times 4B00$.

So for TMR0 to overflow we would need to preload it to 4B00 from the end. So it would overflow in $0 \times 4B00$ pulses.

To do this TMR0 would need to be set to $0xFFFF - 0 \times 4B00 = 0xB4FF$ and the code to do this would be:

```
TMR0H=0xB4;
TMR0L=0xFF;
// TMR0 then overflows and generates an interrupt in 0x4B00 pulses (10 min).
```