Microprocessor Interfacing & Programming

LECTURE 26 & 27

PIC18 Interrupts

- A single microcontroller can serve several devices. There are 2 methods by which devices receive service from the microcontroller:

1. Interrupts
2. Polling

- In the interrupt method, whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal. Upon receiving an interrupt signal, the microcontroller stops whatever it is doing and serves the device.
- The program associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler.
- In polling, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service.

Pros and Cons

- Polling is not the efiicient use, it cannot assign priority because it checks all the devices in a round-robin fashion.
- In interrupt method, the microcontroller can also ignore/mask a device request for service. This is not possible with the polling method.
- The major disadvantage of polling is that it waites much of the microcontroller’s time by polling devices that do not need service.

Interrupt service routine

- For every interrupt, there must be an interrupt service routine (ISR) or interrupt handler.
- When an interrupt is invoked, the microcontroller runs the ISR.
- In most microprocessors, for every interrupt there is a fixed location in memory that hold the address of its ISR.
- The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table.
- In PIC18, there are only two locations for interrupt vector table.

Interrupt	ROM Location (Hex)
Power-on Reset	0000
High Priority Interrupt	0008 (Default upon power-on reset)
Low Priority Interrupt	0018 (See Section 11.6)

Steps in executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps:

1. It finishes the instruction it is executing and saves the address of the next instruction (program counter) on the stack.
2. It jumps to a fixed location in memory called the interrupt vector table. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
3. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETFIE (return from interrupt exit).
4. Upon executing the RETFIE instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

Sources of interrupts in PIC18

1. There is an interrupt set aside for each of the timers.
2. External hardware interrupts. INT0, INT1, INT2 respectively.
3. Serial communication's USART has two interrupts, one for receive and one for transmit.
4. ADC and many more.

How interrupts work in PIC18:

1. Interrupt occurs
2. CPU automatically jumps to the vector at 0x0008 or 0x0018 (in ROM)
3. These locations contain a GOTO to the ISR (also in ROM)
4. CPU executes ISR from ROM

Enabling an interrupt

- Upon reset, all interrupts are disabled (masked).
- The interrupts must be enabled (unmasked) by software in order for the microcontroller to respond to them.
- D7 bit of the INTCON register is responsible for enabling and disabling the interrupts globally.

D7							D0
GIE		TMR0IE	INT0IE				

GIE (Global Interrupt Enable)

GIE = 0 Disables all interrupts. If GIE = 0, no interrupt is acknowledged, even if they are enabled individually.

If GIE = 1, interrupts are allowed to happen. Each interrupt source is enabled by setting the corresponding interrupt enable bit.

TMR0IE Timer0 interrupt enable
= 0 Disables Timer0 overflow interrupt
= 1 Enables Timer0 overflow interrupt

INT0IE Enables or disables external interrupt 0
= 0 Disables external interrupt 0
= 1 Enables external interrupt 0

These bits, along with the GIE, must be set high for an interrupt to be responded to. Upon activation of the interrupt, the GIE bit is cleared by the PIC18 itself to make sure another interrupt cannot interrupt the microcontroller while it is servicing the current one. At the end of the ISR, the RETFIE instruction will make GIE = 1 to allow another interrupt to come in.

PEIE (PEripheral Interrupt Enable)

For many of the peripherals, such as Timers 1, 2, .. and the serial port, we must enable this bit in addition to the GIE bit. (See Figure 11-2.)

Steps in enabling an interrupt

To enable any one of the interrupts, we take the following steps:

1. Bit D7 (GIE) of the INTCON register must be set to HIGH to allow the interrupts to happen. This is done with the "BSF INTCON, GIE" instruction.
2. If GIE = 1, each interrupt is enabled by setting to HIGH the interrupt enable (IE) flag bit for that interrupt. Because there are a large number of interrupts in the PIC18, we have many registers holding the interrupt enable bit. Figure 11-2 shows that the INTCON has interrupt enable bits for Timer0 (TMR0IE) and external interrupt 0 (INT0IE). As we study each of peripherals throughout the book we will examine the registers holding the interrupt enable bits. It must be noted that if GIE = 0, no interrupt will be responded to, even if the corresponding interrupt enable bit is high. To understand this important point look at Example 11-1.
3. As shown in Figures 11-2 and 11-3, for some of the peripheral interrupts such as TMR1IF, TMR2IF, and TXIF, we have to enable the PEIE flag in addition to the GIE bit.

Example

Show the instructions to (a) enable (unmask) the Timer0 interrupt and external hardware interrupt 0 (INT0), and (b) disable (mask) the Timer0 interrupt, then (c) show how to disable (mask) all the interrupts with a single instruction.

Solution:

(a)
```
BSF INTCON,TMR0IE   ;enable(unmask) Timer0 interrupt
BSF INTCON,INT0IE ;enable external interrupt 1(INT0)
BSF INTCON,GIE ;allow interrupts to come in
```

We can perform the above actions with the following two instructions:

```
MOVLW B'10110000'   ;GIE = 1, TMR0IF = 1,INTIF0 = 1
MOVWF INTCON        ;load the INTCON reg
```

(b)
```
BCF INTCON,TMR0IE ;mask (disable) Timer0 interrupt
```

(c)
```
BCF INTCON,GIE      ;mask all interrupts globally
```

Programming Timer Interrupts

Rollover timer flag and interrupt:

- We can use interrupts instead of polling method to avoid tying down the controller.
- If the timer interrupt in the interrupt register is enabled, TMR0IF is raised whenever the timer rolls over and the microcontroller jumps to the interrupt vector table to service the ISR.
- In this way, the microcontroller can do other things until it is notified that the timer has rolled over.

Interrupt	Flag Bit	Register	Enable Bit	Register
Timer0	TMR0IF	INTCON	TMR0IE	INTCON
Timer1	TMR1IF	PIR1	TMR1IE	PIE1
Timer2	TMR2IF	PIR1	TMR2IE	PIE1
Timer3	TMR3IF	PIR3	TMR3IE	PIE2

		TMR0IE			TMR0IF		

- Assume that PORTC is connected to 8 switches and PORTD to 8 LEDs. This program uses Timer0 to generate a square wave on pin RB5, while at the same time the data is being transferred from PORTC to PORTD.

```
     ORG   0000H
     GOTO MAIN        ;bypass interrupt vector table
;--on default all interrupts land at address 00008
     ORG   0008H      ;interrupt vector table
     BTFSS INTCON,TMR0IF   ;Timer0 interrupt?
     RETFIE                ;No. Then return to main
     GOTO  T0_ISR          ;Yes. Then go Timer0 ISR
;--main program for initialization and keeping CPU busy
     ORG   00100H     ;after vector table space
MAIN BCF   TRISB,5    ;PB5 as an output
     CLRF TRISD       ;make PORTD output
     SETF TRISC       ;make PORTC input
```

```
      MOVLW  0x08        ;Timer0,16-bit,
                         ;no prescale,internal clk
      MOVWF  T0CON       ;load T0CON reg
      MOVLW  0xFF        ;TMR0H = FFH, the high byte
      MOVWF  TMR0H       ;load Timer0 high byte
      MOVLW  0xF2        ;TMR0L = F2H, the low byte
      MOVWF  TMR0L       ;load Timer0 low byte
      BCF INTCON,TMR0IF;clear timer interrupt flag bit
      BSF T0CON,TMR0ON       ;start Timer0
      BSF INTCON,TMR0IE      ;enable Timer 0 interrupt
      BSF INTCON,GIE   ;enable interrupts globally
;--keeping CPU busy waiting for interrupt
OVER  MOVFF  PORTC,PORTD ;send data from PORTC to PORTD
      BRA OVER               ;stay in this loop forever
;--------------------------ISR for Timer 0
T0_ISR
      ORG 200H
      MOVLW  0xFF        ;TMR0H = FFH, the high byte
      MOVWF  TMR0H       ;load Timer0 high byte
      MOVLW  0xF2        ;TMR0L = F2H, the low byte
      MOVWF  TMR0L       ;load Timer0 low byte
      BTG   PORTB,5      ;toggle RB5
      BCF INTCON,TMR0IF ;clear timer interrupt flag bit
EXIT  RETFIE ;return from interrupt (See Example 11-2)
      END
```

In the MAIN program, we initialize the Timer0 register and then enter an infinite loop to keep the CPU busy. This could be a real-world application being executed by the CPU. In this case, the loop gets data from PORTC and sends it to PORTD. While the PORTC data is brought in and issued to PORTD continuously, the TMR0IF flag is raised as soon as Timer0 rolls over, and the microcontroller gets out of the loop and goes to 00008H to execute the ISR associated with Timer0. At this point, the PIC18 clears the GIE bit (D7 of INTCON) to indicate that it is currently serving an interrupt and cannot be interrupted again; in other words, no interrupt inside the interrupt. In Section 11.6, we show how to allow an interrupt inside an interrupt.

The ISR for Timer0 is located starting at memory location 00200H because it is too large to fit into address space 08–17H, the address allocated to high-priority interrupts.

In the ISR for Timer0, notice that the "`BCF INTCON, TMR0IF`" instruction is needed before the RETFIE instruction. This will ensure that a single interrupt is serviced once and is not recognized as multiple interrupts.

RETFIE must be the last instruction of the ISR. Upon execution of the RETFIE instruction, the PIC18 automatically enables the GIE (D7 of the INTCON register) to indicate that it can accept new interrupts.

RETURN vs RETFIE

What is the difference between the RETURN and RETFIE instructions? Explain why we cannot use RETURN instead of RETFIE as the last instruction of an ISR.

Solution:

Both perform the same actions of popping off the top bytes of the stack into the program counter, and making the PIC18 return to where it left off. However, RETFIE also performs the additional task of clearing the GIE flag, indicating that the servicing of the interrupt is over and the PIC18 now can accept a new interrupt. If you use RETURN instead of RETFIE as the last instruction of the interrupt service routine, you simply block any new interrupt after the first interrupt, because the GIE would indicate that the interrupt is still being serviced.

Program 11-2 uses Timer0 and Timer1 interrupts to generate square waves on pins RB1 and RB7 respectively, while data is being transferred from PORTC to PORTD.

```
;Program 11-2
            ORG   0000H
            GOTO  MAIN          ;bypass interrupt vector table
      ;--on default all interrupts land at address 00008
            ORG   0008H         ;interrupt vector table
            GOTO  CHK_INT       ;go to an address with more space
      ;--check to see the source of interrupt
            ORG   0040H         ;we got here from 0008
      CHK_INT
            BTFSC INTCON,TMR0IF       ;Is it Timer0 interrupt?
            BRA   T0_ISR              ;Yes. Then branch to T0_ISR
            BTFSC PIR1,TMR1IF         ;Is it Timer1 interrupt?
            BRA   T1_ISR              ;Yes. Then branch to T1_ISR
            RETFIE                    ;No. Then return to main
      ;-main program for initialization and keeping CPU busy
            ORG   0100H  ;somewhere after vector table space
      MAIN  BCF   TRISB,1       ;PB1 as an output
            BCF   TRISB,7       ;PB7 as an output
            CLRF  TRISD         ;make PORTD output
            SETF  TRISC         ;make PORTC input
            MOVLW  0x08         ;Timer0,16-bit,
                                ;no prescale,internal clk
            MOVWF  T0CON        ;load T0CON reg
            MOVLW  0xFF         ;TMR0H = FFH, the high byte
            MOVWF  TMR0H        ;load Timer0 high byte
            MOVLW 0xF2          ;TMR0L = F2H, the low byte
            MOVWF  TMR0L        ;load Timer0 low byte
            BCF INTCON,TMR0IF;clear Timer0 interrupt flag bit
            MOVLW  0x0          ;Timer1,16-bit,
                                ;no prescale,internal clk
            MOVWF  T1CON        ;load T1CON reg
            MOVLW  0xFF         ;TMR1H = FFH, the high byte
            MOVWF  TMR1H        ;load Timer0 high byte
            MOVLW 0xF2          ;TMR1L = F2H, the low byte
            MOVWF  TMR1L        ;load Timer1 low byte
            BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
            BSF INTCON,TMR0IE       ;enable Timer0 interrupt
            BSF PIE1,TMR1IE         ;enable Timer1 interrupt
            BSF INTCON,PEIE  ;enable peripheral interrupts
            BSF INTCON,GIE          ;enable interrupts globally
            BSF T0CON,TMR0ON        ;start Timer0
            BSF T1CON,TMR1ON        ;start Timer1
      ;--keeping CPU busy waiting for interrupt
      OVER  MOVFF PORTC,PORTD ;send data from PORTC to PORTD
            BRA OVER                ;stay in this loop forever
      ;---------------------------ISR for Timer 0
      T0_ISR
            ORG 200H
```

```
        MOVLW  0xFF         ;TMR0H = FFH, the high byte
        MOVWF  TMR0H        ;load Timer0 high byte
        MOVLW 0xF2          ;TMR0L = F2H, the low byte
        MOVWF TMR0L         ;load Timer0 low byte
        BTG   PORTB,1       ;toggle PB1
        BCF INTCON,TMR0IF ;clear timer interrupt flag bit
        GOTO CHK_INT
;--------------------------ISR for Timer1
T1_ISR
        ORG 300H
        MOVLW  0xFF         ;TMR1H = FFH, the high byte
        MOVWF  TMR1H        ;load Timer0 high byte
        MOVLW 0xF2          ;TMR1L = F2H, the low byte
        MOVWF TMR1L         ;load Timer1 low byte
        BTG PORTB,7
        BCF PIR1,TMR1IF ;clear Timer1 interrupt flag bit
        GOTO CHK_INT
        END
```

```
//Program 11-2C (C version of Program 11-2)
#include <p18F458.h>
#define myPB1bit PORTBbits.RB1
#define myPB7bit PORTBbits.RB7

void T0_ISR(void);
void T1_ISR(void);

#pragma interrupt chk_isr  //used for high-priority
                           //interrupt only
void chk_isr (void)
{
     if (INTCONbits.TMR0IF==1)  //Timer0 causes interrupt?
          T0_ISR();             //Yes. Execute Timer0 ISR
     if(PIR1bits.TMR1IF==1)     //Or was it Timer1?
          T1_ISR();             // Yes. Execute Timer1 ISR
}
```

```
#pragma code
void main(void)
  {
    TRISBbits.TRISB1=0;      //RB1 = OUTPUT
    TRISBbits.TRISB7=0;      //RB7 = OUTPUT
    TRISC = 255;             //PORTC = INPUT
    TRISD = 0;          //PORTD = OUTPUT
    T0CON=0x0;          //Timer 0, 16-bit mode, no prescaler
    TMR0H=0x35;         //load TH0
    TMR0L=0x00;         //load TL0
    T1CON=0x88;         //Timer 1, 16-bit mode, no prescaler
    TMR1H=0x35;         //load TH1
    TMR1L=0x00;         //load TL1
    INTCONbits.TMR0IF=0;     //clear TF0
    PIR1bits.TMR1IF=0;       //clear TF1
    INTCONbits.TMR0IE=1;     //enable Timer0 interrupt
    INTCONbits.TMR0IE=1;     //enable Timer1 interrupt
    T0CONbits.TMR0ON=1;      //turn on Timer0
    T1CONbits.TMR1ON=1;      //turn on Timer1
    INTCONbits.PEIE=1;//enable all peripheral interrupts
    INTCONbits.GIE=1; //enable all interrupts globally
   while(1)     //keep looping until interrupt comes
    {
      PORTD=PORTC;    //send data from PORTC to PORTD
    }
  }

void T0_ISR(void)
  {
    myPB1bit=~myPB1bit;      //toggle PORTB.1
    TMR0H=0x35;              //load TH0
    TMR0L=0x00;              //load TL0
    INTCONbits.TMR0IF=0;     //clear TF0
  }

void T1_ISR(void)
  {
    myPB7bit=~myPB7bit;      //toggle PORTB.7
```

```
    TMR1H=0x35;              //load TH0
    TMR1L=0x00;              //load TL0
    PIR1bits.TMR1IF=0;       //clear TF1
  }
```

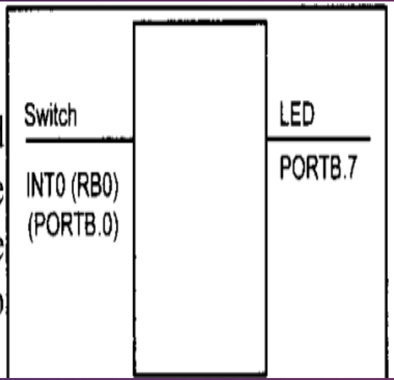
External Hardware Interrupts

- PIC18 has 3 external hardware interrupts.
- Pins RB0, RB1 and RB2 are designated as INT0, INT1 and INT2.
- Upon activation of these pins, PIC18 gets interrupted in whatever it is doing and jumps to the vector table to perform ISR.
- They must be enabled before they can take effect.

Interrupt (Pin)	Flag bit	Register	Enable bit	Register
INT0 (RB0)	INT0IF	INTCON	INT0IE	INTCON
INT1 (RB1)	INT1IF	INTCON3	INT1IE	INTCON3
INT2 (RB2)	INT2IF	INTCON3	INT2IE	INTCON3

- Upon power-on reset, PIC18 makes INT0, INT1 and INT2 rising (positive) edge-triggered interrupts.

Program 11-4 connects a switch to INT0 and an LED to pin RB7. In this program, every time INT0 is activated, it toggles the LED, while at the same time data is being transferred from PORTC to PORTD.



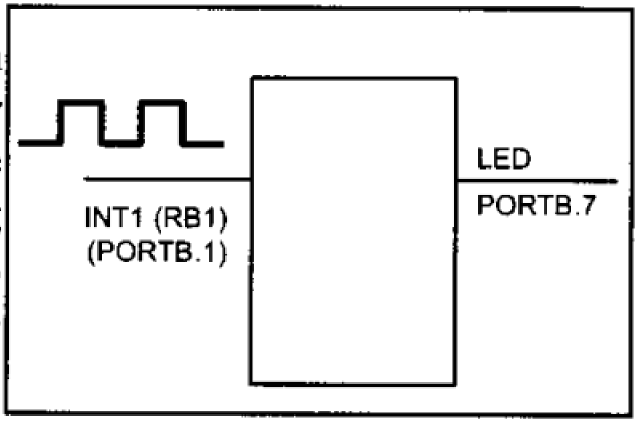

```
      ORG   0000H
      GOTO  MAIN                ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
      ORG   0008H               ;interrupt vector table
      BTFSS INTCON,INT0IF       ;Did we get here due to INT0?
      RETFIE                    ;No. Then return to main
      GOTO  INT0_ISR            ;Yes. Then go INT0 ISR
;;--the main program for initialization
      ORG   00100H
MAIN  BCF   TRISB,7             ;PB7 as an output
      BSF TRISB,INT0            ;make INT0 an input pin
      CLRF TRISD                ;make PORTD output
      SETF TRISC                ;make PORTC input
      BSF  INTCON,INT0IE        ;enable INT0 interrupt
      BSF  INTCON,GIE           ;enable interrupts globally
OVER  MOVFF PORTC,PORTD         ;send data from PORTC to PORTD
      BRA OVER                  ;stay in this loop forever
;---------------------------ISR for INT0
INT0_ISR
      ORG 200H
      BTG PORTB,7               ;toggle PB7
      BCF  INTCON,INT0IF        ;clear INT0 interrupt flag bit
      RETFIE                    ;return from ISR
      END
```

Negative edge-triggered interrupts

	INTEDG0	INTEDG1	INTEDG2				

INTEDGx External Hardware Interrupt Edge trigger bit
0 = Interrupt on negative (falling) edge
1 = Interrupt on positive (rising) edge (Default for power-on reset)

In Program 11-5 we assume that pin RB1 (INT1) is connected to a pulse generator and the pin RB7 is connected to an LED. The program will toggle the LED on the falling edge of the pulse. In other words, the LED is turned on and off at the same rate as the pulses are applied to the INT1 pin.



```
        ORG    0000H
        GOTO  MAIN          ;bypass interrupt vector table
;--on default all interrupts go to to address 00008
        ORG    0008H        ;interrupt vector table
        BTFSS INTCON3,INT1IF  ;Did we get here due to
                              ;INT1 interrupt?
        RETFIE                ;No. Then return to main
        GOTO   INT1_ISR       ;Yes. Then go INT1 ISR
;--the main program for initialization
        ORG    00100H
MAIN    BCF    TRISB,7      ;PB7 as an output
        BSF    TRISB,INT1   ;make INT1 an input pin

        BSF   INTCON3,INT1IE   ;enable INT1 interrupt
        BCF INTCON2,INTEDG1    ;make it negative
                               ;edge-triggered
        BSF INTCON,GIE         ;enable interrupts globally
OVER    BRA OVER               ;stay in this loop forever
;-------------------------ISR for INT1
INT1_ISR
        ORG 200H
        BTG PORTB,7            ;toggle on RB7
        BCF INTCON3,INT1IF ;clear INT1 interrupt flag bit
        RETFIE
        END
```