

# Using Interrupts

# Objectives

- Exam source code and comment ( all source C and is provided)
- Learn usage of interrupts by examining Interrupt based code examples using PIC16F887 peripherals
  - Case Studies
    - A. Timer0 overflow interrupt usage
    - B. Timer1 overflow interrupt usage
    - C. ADC conversion interrupt usage
    - D. RBO button Interrupt usage

# In general

- It is up to the user to decide when to use interrupts.
- By deciding not to use interrupts the user may run the risk of missing response to critical outside events.
  - Ie. Air-bag, Anti-lock breaking, speed control
- PIC interrupts have guaranteed latency time for response
- Interrupt increase the response time of the processor to more “real-time” versus the alternative of “polling” peripherals.
- In interrupt the peripheral tells the processor when it needs service immediately

# Compiler Interrupt Support

- The compiler incorporates features allowing interrupts to be handled from C code. Interrupt functions are often called interrupt service routines (ISR).
- The function qualifier **interrupt** may be applied to any number of C function definitions to allow them to be called directly from the hardware interrupts.
- An example of an interrupt function for a midrange PIC processor is shown here.

```
int tick_count;
void interrupt tc_int(void)
{
if (TOIE && TOIF) {
TOIF=0;
++tick_count;
}}
```

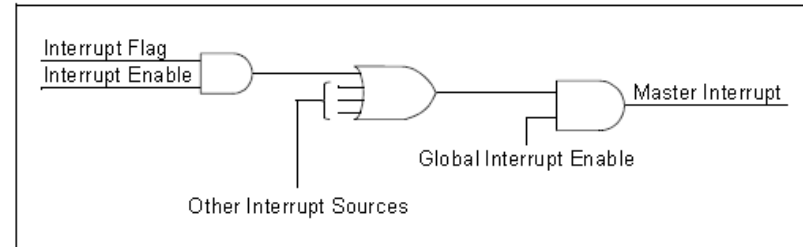
# Mid-range PIC Specifics

- C Code has one interrupt function that all peripherals that are interrupt enabled shared.
- Interrupt function must determine the source of interrupts ( if using more then one peripheral)
  - It does this by examining all relevant interrupt flag bits XXIF to see which are set
- The interrupt function must clear the source of interrupt before exiting to re-enable that peripheral to get interrupt service
  - It does this by clearing the XXIF bit
- All processer “context saving” and “context restore” are performed automatically for W by interrupt function .

# Interrupts

- **Interrupt Sources**
- Most of the peripherals can generate an interrupt, and some of the I/O pins may be configured to generate an interrupt when they change state.
- When a peripheral needs service or an event occurs, it sets its interrupt flag.
- Each interrupt flag is ANDed with its enable bit and then these are ORed together to form a Master Interrupt.
- This master interrupt is ANDed with the Global Interrupt Enable (GIE).
- The enable bits allow the PIC microcontroller to limit the interrupt sources to certain peripherals.

FIGURE 3-9: INTERRUPT LOGIC SIMPLIFIED



# To Enable Interrupts for TIMER0 and PORTB

- Must set global enable bit (GIE) in INTCON
- For peripherals referenced in INTCON
  - RB0, RB port change, Timer0 set the corresponding Interrupt enable bit and clear the corresponding XXIF bit.
  - Make sure interrupt function contains correct service for the enable peripheral

REGISTER 2-3: INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh OR 18Bh)

	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
	GIE	PEIE	T0IE	INTE	RBIE	T0IF	RBIF
bit 7							bit 0

bit 7	<b>GIE:</b> Global Interrupt Enable bit 1 = Enables all unmasked interrupts 0 = Disables all interrupts
bit 6	<b>PEIE:</b> Peripheral Interrupt Enable bit 1 = Enables all unmasked peripheral interrupts 0 = Disables all peripheral interrupts
bit 5	<b>T0IE:</b> TMR0 Overflow Interrupt Enable bit 1 = Enables the TMR0 interrupt 0 = Disables the TMR0 interrupt
bit 4	<b>INTE:</b> RB0/INT/SEG0 External Interrupt Enable bit 1 = Enables the RB0/INT/SEG0 external interrupt 0 = Disables the RB0/INT/SEG0 external interrupt
bit 3	<b>RBIE:</b> PORTB Change Interrupt Enable bit <sup>(1)</sup> 1 = Enables the PORTB change interrupt 0 = Disables the PORTB change interrupt
bit 2	<b>T0IF:</b> TMR0 Overflow Interrupt Flag bit <sup>(2)</sup> 1 = TMR0 register has overflowed (must be cleared in software) 0 = TMR0 register did not overflow
bit 1	<b>INTF:</b> RB0/INT/SEG0 External Interrupt Flag bit 1 = The RB0/INT/SEG0 external interrupt occurred (must be cleared in software) 0 = The RB0/INT/SEG0 external interrupt did not occur
bit 0	<b>RBIF:</b> PORTB Change Interrupt Flag bit 1 = When at least one of the PORTB <5:0> pins changed state (must be cleared in software) 0 = None of the PORTB <7:4> pins have changed state

**Note 1:** IOCB register must also be enabled.

**2:** T0IF bit is set when Timer0 rolls over. Timer0 is unchanged on Reset and should be initialized before clearing T0IF bit.

# Timer0 Interrupt

- Uses Timer0 interrupt to increment PORTD LEDs
  - Timer0 is a counter implemented in the processor. Under interrupt when its count rolls over it sets a Interrupt flag indicating that it is requesting service from CPU as a result of this event.
  - The CPU clears the interrupt flag ( allowing the counter to continue to another interrupt in the future) and also uses this event to increment PORTD
  - There is no need to do a software based delay.

# INTCON

REGISTER 2-3: INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh OR 18Bh)

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	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
							bit 0
							bit 7

The INTCON register is a readable and writable register, which contains the various enable and flag bits for TMR0 register overflow, PORTB change and external RB0/INT/SEG0 pin interrupts.

bit 7	<b>GIE:</b> Global Interrupt Enable bit 1 = Enables all unmasked interrupts 0 = Disables all interrupts
bit 6	<b>PEIE:</b> Peripheral Interrupt Enable bit 1 = Enables all unmasked peripheral interrupts 0 = Disables all peripheral interrupts
bit 5	<b>TOIE:</b> TMR0 Overflow Interrupt Enable bit 1 = Enables the TMR0 interrupt 0 = Disables the TMR0 interrupt
bit 4	<b>INTE:</b> RB0/INT/SEG0 External Interrupt Enable bit 1 = Enables the RB0/INT/SEG0 external interrupt 0 = Disables the RB0/INT/SEG0 external interrupt
bit 3	<b>RBIE:</b> PORTB Change Interrupt Enable bit <sup>(1)</sup> 1 = Enables the PORTB change interrupt 0 = Disables the PORTB change interrupt
bit 2	<b>TOIF:</b> TMR0 Overflow Interrupt Flag bit <sup>(2)</sup> 1 = TMR0 register has overflowed (must be cleared in software) 0 = TMR0 register did not overflow
bit 1	<b>INTF:</b> RB0/INT/SEG0 External Interrupt Flag bit 1 = The RB0/INT/SEG0 external interrupt occurred (must be cleared in software) 0 = The RB0/INT/SEG0 external interrupt did not occur
bit 0	<b>RBIF:</b> PORTB Change Interrupt Flag bit 1 = When at least one of the PORTB <5:0> pins changed state (must be cleared in software) 0 = None of the PORTB <7:4> pins have changed state

**Note 1:** IOCB register must also be enabled.

**2:** TOIF bit is set when Timer0 rolls over. Timer0 is unchanged on Reset and should be initialized before clearing TOIF bit.

# Timer0 Interrupt Exercise

1. Navigate to: C:\EET250\CS2117 course\Lesson 10 Interrupts with Demo Board\timer0
2. Open interrupt.mcp
3. Select debugger-> PICKIT2
4. Make sure output window says PICKIT2 ready
5. This code uses Timer0 to cause delay for PORTD increment
6. Examine code to understand TIMER0 setting
  - Clock source, prescaler and count
  - What is the calculated delay?
7. Examine code to understand how Timer0 interrupt is set-up and serviced
  - Where is Timer0 interrupt enabled?
  - Where is Global Interrupt enable set?
  - Where does the interrupt service routine reside?
  - What does the clearing of the Timer0 interrupt flag achieve?
8. Build/download/program
9. Validate count operation
10. Set breakpoint at function ISR code line `T0IF = 0;`. Run code, break then run code again. Notice how the ISR is working and is functioning for counter update.
11. Select debugger -> simulator
12. Make sure that simulation is running at 4Mhz clock rate
13. Keep break point at ISR Use stopwatch. Reset and run simulation.
14. At break clear stopwatch and then resume simulation
15. At next break the stopwatch will show measured time delay—is it the same as your calculations?

# To Enable Interrupts for TIMER1 ADC and other peripherals

- Must set global enable bit (GIE) in INTCON
- For peripherals outside of those referenced in INTCON set bit 6 EEIE bit general interrupt enable for other peripherals
- Then go to specific interrupt enable either in PIE1 or PIE2 register
  - Interrupt function then must check pending request from PIR1 or PIR2 as needed.

TABLE 16-6: SUMMARY OF INTERRUPT REGISTERS

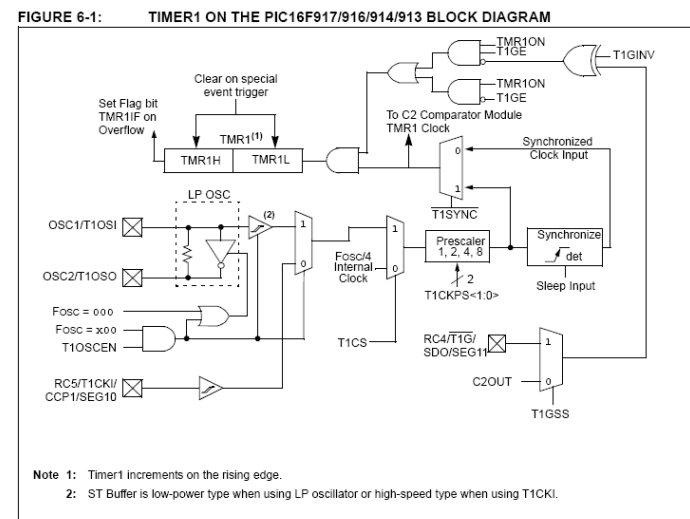
Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
0Bh, 8Bh	INTCON	GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000x
0Ch	PIR1	EEIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
0Dh	PIR2	OSFIF	C2IF	C1IF	LCDIF	—	LVDIF	—	CCP2IF	0000 -0-0	0000 -0-0
8Ch	PIE1	EEIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
8Dh	PIE2	OSFIE	C2IE	C1IE	LCDIE	—	LVDIE	—	CCP2IE	0000 -0-0	0000 -0-0

# Timer1 Interrupt

- Uses Timer1 interrupt to increment PORTD LEDs
  - Timer1 is a counter implemented in the processor. Under interrupt when its count rolls over it sets a Interrupt flag indicating that it is requesting service from CPU as a result of this event.
  - The CPU clears the interrupt flag ( allowing the counter to continue to another interrupt in the future) and also uses this event to increment PORTD
  - There is no need to do a software based delay.

# Tmer1 Interrupt

- Timer 1 is a 16 bit timer
- Setting:
  - T1CON
    - Set up and prescaler
  - PIR1
    - Rollover flag for Timer1
  - 16 bit counter
    - TMR1L
    - TMR1H
- Interrupts
  - The Timer1 register pair (TMR1H:TMR1L) increments to FFFFh and rolls over to 0000h.
  - When Timer1 rolls over the Timer1 Interrupt Flag bit (PIR1<0>) is set.
  - To enable the interrupt on rollover, you must set these bits:
    - Timer1 Interrupt Enable bit (PIE1<0>) PEIE bit (INTCON<6>) and GIE bit (INTCON<7>)
  - The interrupt is cleared by clearing the TMR1IF bit in the Interrupt Service Routine.



# Timer1 Interrupt Exercise

1. Navigate to C:\EET250\CS2117 course\Lesson 10 Interrupts with Demo Board\timer1int
2. Open cPKLEDint.mcp
3. Select debugger-> PICKIT2
4. Make sure output window says PICKIT2 ready
5. This code uses Timer0 to cause delay for PORTD increment
6. Examine code to understand TIMER0setting
  - Clock source, prescaler and count
  - What is the calculated delay?
7. Examine code to understand how Timer0 interrupt is set-up and serviced
  - Where is Timer1 interrupt enabled?
  - Why is PEIE set?
  - Where is Global Interrupt enable set?
  - Where does the interrupt service routine reside?
  - What does the clearing of the Timer1 interrupt flag achieve?
8. Build/download/program
9. Validate count operation
10. Set breakpoint at function ISR code line `TMR1IF = 0;`. Run code, break then run code again. Notice how the ISR is working and is functioning for counter update.
11. Select debugger -> simulator
12. Make sure that simulation is running at 4Mhz clock rate
13. Keep break point at ISR Use stopwatch. Reset and run simulation.
14. At break clear stopwatch and then resume simulation
15. At next break the stopwatch will show measured time delay—is it the same as your calculations?

# ADC Interrupt

- Uses ADC interrupt to retrieve digitized pot value and display it using PORTD LEDs
  - ADC will interrupt when a conversion is complete and the ADC value is ready for retrieval.
  - The CPU clears the interrupt flag ( allowing the ADC to continue to another interrupt in the future), initiates another conversion and also uses this event to increment PORTD
  - There is no need to do a software based state machine to start conversion and wait on result.

# ADC interrupt Exercise

1. Navigate to: C:\EET250\CS2117 course\Lesson 10 Interrupts with Demo Board\ADCInterrupt
2. open ADCInterrupt.mcp
3. Examine code.
4. The ADC conversion settings should be similar to other non-interrupts versions we worked with.
5. Notice that the conversion occurs as part of the interrupt—there is no need for an ADC state machine
6. What SFR bits are used to enable interrupts?
7. Notice that besides interrupt enabling an initial conversion must be initiated?
8. What does the interrupt service routine do? How are ADC conversions maintained?
9. Build code and download to PICKIT2
10. Execute. Exercise pot changes. Does this application perform in a similar way to earlier non-interrupt ADC.mcp?

# Interrupts exercises so far

- Not very exciting
- Just replacing an polled non-interrupt version with an interrupt version with a “do-Nothing” continuous loop
- Interrupt really afford the user with the ability to almost do several things at one
- The next example InterruptButton.mcp will illustrate this

# RB0 Interrupt

- Part of this next exercise is to use INTCON interrupt RB0 .
  - This interrupt occurs when RB0 changes state.
  - RB0 is hooked on the demo board to a pushbutton. This is an ideal solution to respond effectively to a asynchronous event like a user pushbutton activation while the processor is busy doing something else.
- Our exercise will have the processor blinking one led RD7 LED during main loop and when the RB0 interrupt occurs it will toggle RD0 LED

# Two Simultaneous operations

- InterruptButton.mcp uses the RB0 interrupt to toggle RD7 LED while main loop is toggling RD0 led without interfering with each other

```
void interrupt rb0int(void)
{
    //wait 20 msec
    j=0;
    while(j<TwoHundredms)
    { j++; }
    INTF = 0; // Reset RB0 Interrupt Request

    RD7 = RD7 ^ 1; // Toggle RD7 to Turn ON/OFF LED under interrupt
} // rb0int

main()
{
    RD0 =0;
    RD7 =0;
    TRISB0 =1; //rb0 is an input

    PORTD =0x00;
    TRISD7 = 0; // MSB LED
    TRISD0 =0; // LSB LED
    GIE = 1; // Enable Global Interrupts
    INTE = 1; // Enable RB0 Interrupt

    //main loop just delay 600 ms and toggle LED 0
    while (1)
    {
        i =0;
        while(i<TwoHundredms)
        { i++; }
        RD0 = RD0 ^ 1; // Toggle RD0 to Turn ON/OFF LED
    }
} // End cDebounce
```

# RB0 interrupt Exercise

1. Navigate to C:\EET250\CS2117 course\Lesson 10 Interrupts with Demo Board\InterruptButton
2. open InterruptButton.mcp
3. Examine code.
4. What SFR bits are used to enable interrupts?
5. What happens during RB0 interrupt service?
6. Describe the loop being executed during main  
Build code and download to PICKIT2
7. Execute. Exercise button pushes. Does this interrupt interfere with the main blinking application?