What is an Interrupt?

- A transfer of program control triggered by some event
- Like a subroutine call, except caused by some event or action external to the program rather than an explicit “call” instruction
- An interrupt causes the processor to temporarily suspend whatever program it is executing in order to “service the interrupt”.
  - The “interrupt service routine” is pretty much like a subroutine
  - Return address is automatically saved on stack
  - When ISR is completed, program control is returned to the point where the interrupt occurred
Interrupt Processing

"Main" Program:

- Instr i
- Instr i+1
- Instr i+2
- Instr i+3
- Instr i+4
- Instr i+5

ISR:

- Instr 1
- Instr 2
- Instr i+1
- Instr i+2
- Instr i+3
- Instr i+4
- Instr i+5

retfie

"Main" Program:

Suppose an interrupt occurs here

return address saved on stack

ISR:

- Instr 1
- Instr 2
- Instr i+1
- Instr i+2
- Instr i+3
- Instr i+4
- Instr i+5

retfie
Interrupt Processing

"Main" Program:

Instr i
Instr i+1
Instr i+2
Instr i+3
Instr i+4
Instr i+5

ISR:

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5

return address popped to PC

refie

"Main" Program:

Instr i
Instr i+1
Instr i+2
Instr i+3
Instr i+4
Instr i+5

ISR:

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5

refie

return address popped to PC

Interrupt Processing

"Main" Program:

Instr i
Instr i+1
Instr i+2
Instr i+3
Instr i+4
Instr i+5

ISR:

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5

refie

"Main" Program:

Instr i
Instr i+1
Instr i+2
Instr i+3
Instr i+4
Instr i+5

ISR:

Instr 1
Instr 2
Instr 3
Instr 4
Instr 5

refie
PIC18F452 Interrupts

- Multiple Interrupt Sources
  - Timers (timer rollover)
  - On-chip peripheral modules
    - USART
    - A/D converter
    - CCP module
    - etc.
- External Interrupts requests
  - Pins RB0-RB2
  - Change on Pin RB4-RB7

Why Are Interrupts Useful

- Consider Lab 3
  - Need to time multiple intervals:
    - 10 msec. interval—read and process RPG input
    - Duty-cycle intervals (on-time, off-time)—toggle RC2 to attain proper duty cycle
      - 99.9 msec. \(\geq\) on-time \(\geq\) 0.1 msec.
    - Duty-cycle on-time interval (off-time interval) interval can be much shorter than the 10 msec. RPG interval.

Main Loop Structure for Lab 3

1. Read RPG and compute new duty-cycle intervals (on-time, off-time)
2. current duty-cycle interval done?
   - no
   - Toggle RC2 output
   - Reinitialize duty-cycle timer for other interval
   - 10 msec timer expired?
     - no
     - Reinitialize 10 msec timer
     - yes
6. Reinitialize 10 msec timer
Main Loop Structure for Lab 3

- Read RPG and compute new duty-cycle intervals (on-time, off-time)
- 10 msec. timer expired?
- no
- Toggle RC2 output
- 10 msec timer expired?
- no
- Reinitialize duty-cycle timer to other interval
- yes
- Reinitialize 10 msec timer

This doesn’t work since duty cycle interval may be much less than 10 msec.

10 msec. timing loop

Alternative Structure for Lab 3

- Toggle RC2 output
- 10 msec timer expired?
- no
- Reinitialize 10 msec timer
- yes
- Read RPG and compute new duty-cycle intervals
- Duty-cycle interval timer expired?
- no
- Reinitialize duty-cycle timer to time other interval
- yes

This works with respect to the minimum duty cycle interval (0.1 ms) but duty-cycle intervals > 10 msec will mess with the RPG read rate.

Loop iteration time determined by duty-cycle on-time/off-time
Another idea: Use a fixed 0.1 msec. timing loop (min. duty-cycle interval) and use counters (not timers) to time the 10 msec. and duty-cycle intervals.

Decrement 10 msec counter and duty-cycle interval counter

If 10 msec. counter expired?

Yes

Reset 10 msec counter

No

Read RPG and compute new duty cycle intervals

If duty-cycle interval counter expired?

Yes

Toggle RC2 output

No

Reset duty-cycle counter for other fraction of the period

If 0.1 msec. timer expired?

Yes

Reset 0.1 msec timer t

No

0.1 msec timing loop

What if the problem specification required imposing the duty-cycle on a 100 Hz square wave rather than a 10 Hz one?

Decrement 10 msec counter and duty-cycle interval counter

If 10 msec. counter expired?

Yes

Reset 10 msec counter

No

Read RPG and compute new duty cycle intervals

If duty-cycle interval counter expired?

Yes

Toggle RC2 output

No

Reset duty-cycle counter for other fraction of the period

If 0.1 msec. timer expired?

Yes

Reset 0.1 msec timer t

No

0.1 msec timing loop

For a 0.1 msec. timing loop, the loop body can contain at most 250 instruction cycles.

This works, but...all the code in one iteration of the main loop must execute within the minimum duty-cycle interval (0.1 msec).

What if the problem specification required imposing the duty-cycle on a 100 Hz square wave rather than a 10 Hz one?

Now the minimum duty cycle interval is:

0.01 msec (10 microseconds)
How Can We Modify the Design to Control the Duty Cycle of a Higher Frequency Waveform?

- Use interrupts
- General Approach:
  - Configure the duty-cycle timer to generate an interrupt at each roll-over
  - The Interrupt Service Routine can toggle the RC2 output and reinitialize the duty-cycle timer to time the other portion of the waveform period.
  - The RPG code can utilize a 10 msec. timing loop:

New Design Using Interrupts

"Main" program loop

- Read RPG and compute new duty-cycle intervals (on-time, off-time)
- Toggle RC2 output
- Reinitialize 10 msec timer

Interrupt Service Routine

- 10 msec timer expired?
  - Yes: Reinitialize duty-cycle timer
  - No: return
Priority Interrupts

• Interrupt Priorities
  – Many microcontrollers and microprocessors support multiple interrupt priorities
  – Higher priority interrupt requests and interrupt the servicing of lower priority interrupts, but not vice versa.
  – The PIC has a simple, two-level priority scheme

Interrupt Priority

Servicing of second request is deferred until first servicing of first request is complete. i.e. During the processing of a LoPri ISR, further LoPri interrupts are temporarily disabled.

High Priority Interrupt request preempts servicing of low priority ISR. (While servicing a high priority interrupt, all further interrupts—both high and low priority—are temporarily disabled.)
Interrupt Vectors

• In the PIC18F452:
  – High priority interrupts “vector through” program memory location 8
  – Low priority interrupts “vector through” program memory location 18
  – Resets (which are a special form of interrupt) vector through program memory location 0.

This simply means that, when a high priority interrupt is serviced, the program counter is loaded with the address 8 (after the return address has been pushed onto the stack).

Typical Code to Set up Vectors

```
org  0x0000             ;Reset vector
goto Mainline
org  0x0008             ;High priority interrupt vector
goto HiPriISR
org  0x0018             ;Low priority interrupt vector
goto LoPriISR
```

Note: there is only one High Priority ISR and one Low-Priority ISR. If there is more than one potential source of interrupts at a given priority level, the ISR must POLL to determine which source caused the interrupt.
ISR Example

Interrupt Service Routine

LowPriISR

Toggle RC2 output

Reinitialize half-period timer

return

ISR Example

Interrupt Service Routine

LowPriISR

blg PORTC, RC2 ; toggle RC2
rcall ReinitHPtimer ; reinitialize ; half-period timer
retfie ; return from interrupt

Note: This subroutine must clear the timer interrupt flag.

ISR Example

Interrupt Service Routine

LowPriISR

Toggle RC2 output

Reinitialize duty-cycle timer to time other fraction of the waveform period

return

ISR Example

Interrupt Service Routine

LowPriISR

blg PORTC, RC2 ; toggle RC2
rcall ReinitDCtimer ; reinitialize ; duty-cycle timer
retfie ; return from interrupt

Polling Example

Suppose that both the TenMsecTimer and DCtimer are configured to generate low priority interrupts

Main program loop:

Do Nothing

Road RC2 output

TenMsecTimer expired

Which Timer Expired?

DCtimer expired

Reinitialize Tenmsec timer

Reinitialize duty-cycle timer

return

Note: The instruction used to return from an interrupt. In addition to restoring the PC from the stack, it re-enables interrupts at this priority level. When a low (high) priority interrupt occurs, further low (any priority) interrupts are disabled. The retfie re-enables the interrupts.
Polling Example
Suppose that both the TenMsecTimer and DCtimer are configured to generate low priority interrupts.

Main program loop:
- Do Nothing
- WhichTimer expired?
  - TenMsecTimer expired
    - Read RPG and update duty-cycle intervals
    - Toggle RC2 output
    - Reinitialize 10msec timer
    - Reinitialize duty-cycle timer
  - HPtimer expired
    - Read RPG and update duty-cycle intervals
    - Reinitialize 10msec timer
    - Toggle RC2 output
    - return

Note: This is no better than the original timing-loop based approach. Waveform frequency is limited by the execution time of the main program loop.

Polling Example—Better Approach
Configure the TenMsecTimer to generate low priority interrupts and the DCcounter to generate high priority interrupts.

Main program loop:
- Do Nothing
- WhichTimer expired?
  - TenMsecTimer expired
    - Read RPG and update duty-cycle intervals
    - Toggle RC2 output
    - Reinitialize 10msec timer
    - Reinitialize duty-cycle timer
    - return
  - HPtimer expired
    - Read RPG and update duty-cycle intervals
    - Reinitialize 10msec timer
    - Toggle RC2 output
    - return

Note: Performance-wise, this is no improvement over the earlier structure using a 10 msec main timing loop (without interrupts) and an ISR to handle RC2 toggling—i.e. no real advantage to using interrupts for both timers (unless there is something else important for the main program to do.)

ISRs—Saving and Restoring Registers
- The point of occurrence of interrupts is not predictable
- An ISR may alter the state of the processor
  - e.g. change the value of status bits, WREG, BSR, etc
- This could easily mess up program behavior

Main program:
- movf TABLAT, F
- bnz loop
- 
- ISR
- incf COUNTER, F
- retfie

Suppose an interrupt occurs here...

ISR
- movf TABLAT, F
- bnz loop
- 
- ISR
- incf COUNTER, F
- retfie
ISRs—Saving and Restoring Registers

- The point of occurrence of interrupts is not predictable
- An ISR may alter the state of the processor
  - e.g. change the value of status bits, WREG, BSR, etc
- This could easily mess up program behavior

Main program:

```
movf TABLAT, F
bnz loop
```

ISR

```
incf COUNTER, F
retfie
```

Suppose an interrupt occurs here

ISR may change the value of the Z status bit

Order is important here. Why??

ISRs—Saving and Restoring Registers

- Generally a good idea to save critical registers (STATUS, WREG, possibly BSR and/or FSR0-FSR2) at the entry point to an ISR and restore them right before the return

LoPriISR

```
movff STATUS, STATUS_TEMP    ; save STATUS in temp location
movwf WREG_TEMP                       ; save WREG in temp location
movf TABLAT, F
bnz loop
movf WREG_TEMP, W                  ; restore WREG
movff STATUS_TEMP, STATUS     ; restore STATUS
retfie ; return from interrupt
```

ISR changes the value of WREG

LoPriISR

```
movff STATUS, STATUS_TEMP    ; save STATUS in temp location
movwf WREG_TEMP                       ; save WREG in temp location
movw 0x40
movwf PORTB
movlw high Bignum
retfie
```

ISR may change the value of the WREG

Order is important here. Why??

ISRs—Saving and Restoring Registers

- Generally a good idea to save critical registers (STATUS, WREG, possibly BSR and FSR0-FSR2) at the entry point to an ISR and restore them right before the return

LoPriISR

```
movff STATUS, STATUS_TEMP    ; save STATUS in temp location
movwf WREG_TEMP                       ; save WREG in temp location
movff STATUS_TEMP, STATUS     ; restore STATUS
retfie ; return from interrupt
```
ISR—Saving and Restoring Registers

- The PIC provides a "fast" way to save and restore critical registers during interrupt servicing
  - When an interrupt occurs, the values of WREG, STATUS, and BSR are automatically saved in "shadow registers"
  - The return instruction: retfie FAST
    will cause WREG, STATUS and BSR to be restored from the shadow registers
- Note: This is only reliable for high-priority interrupts since if a low-priority interrupt is preempted by a high priority one, its saved register values will be overwritten by those of the high priority interrupt
- In general, low priority ISRs should save and restore registers manually

Critical Regions

- Certain sequences of instructions must be executed in an indivisible fashion
  - i.e. want to make sure that an interrupt doesn’t happen while this sequence of instructions is being executed
  - Such a sequence of instructions is called a critical region

Critical Region Example

- Suppose an interrupt occurs here
**Critical Region Example**

**Main program**

Read RPG and update duty-cycle intervals

Calculate the (16 bit) count value to be loaded into DCtimer to time the new duty-cycle interval

DC_OnTimeHi ← high order portion of new value

DC_OnTimeLo ← low order portion of new value

Reinitialize DCtimer using the count value stored in DC_OnTimeHi:DC_OnTimeLo

Note: Partially updated count value gets loaded into the timer

**ISR**

loggle RC2

return

**Critical Regions—Another Example**

--- LoopTime subroutine ---

Bignum equ 65536-25000+12+2

LoopTime

btfss INTCON,TMR0IF ;Wait until ten milliseconds are up
bra LoopTime

movff INTCON,INTCONCOPY ;Disable all interrupts to CPU

bcf INTCON,GIEH

movff TMR0L,TMR0LCOPY ;Read 16-bit counter at this moment

movff TMR0H,TMR0HCOPY

movlw low Bignum

addwf TMR0LCOPY,F

movlw high Bignum

addwf TMR0HCOPY,F

movff TMR0HCOPY,TMR0H

movff TMR0LCOPY,TMR0L

movf INTCONCOPY,W ;Restore GIEH interrupt enable bit

andlw B’10000000’

iorwf INTCON,F

bcf INTCON,TMR0IF ;Clear Timer0 flag

return

This is a critical region since precise timing depends on exactly 12 elapsed instruction cycles

Disabling all interrupts prior to the code section and reenabling them afterwards insures the atomicity of the critical region
Implementing Critical Regions

Typically, critical regions are implemented by temporarily disabling interrupts.

- Sometimes it is necessary to disable ALL interrupt sources.
- Other times, it may be sufficient to disable a single interrupt source.

Critical Regions—Concluding Thoughts (For Now)

- Proper enforcement of critical regions can be very important in applications involving interrupts and/or other forms of concurrency.
- Critical regions constitute one of the biggest and trickiest sources of errors in embedded systems.
- We will have much more to say about this later in the semester.

Interrupts—Now for the Gory Details

- Overall Enabling and Disabling of Interrupts—Controlled by INTCON Register: Bits 7-6.
- High order bit of RCON Register (IPEN) selects between regular (two level priority) interrupt scheme or a compatibility mode with earlier PIC16Cxxx processor.
  - Be careful, the default (reset) setting for IPEN is zero, which disables priority interrupts.
  - Must remember to set IPEN as part of the initialization process.

Enabling/Disabling Interrupts
Enabling/Disabling Interrupts

- With one exception, each interrupt source is associated with three important bits (spread out over 10 different SFRs)
  - Priority Bit: controls whether this interrupt source generates a high or low priority interrupt
  - Local Enable Bit: Individually controls whether this interrupt source is enabled or disabled
    - in order for an interrupt source to be enabled, both the local and global enable bits must be set
  - Local Flag Bit: Set when interrupt-generating condition has occurred. Must be manually cleared

### Register Bits for Each Interrupt Source

<table>
<thead>
<tr>
<th>Name</th>
<th>Priority Bit</th>
<th>Local Enable Bit</th>
<th>Local Flag Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT0 external interrupt</td>
<td>INTOCON.INT0IP</td>
<td>INTOCON.INT0IE</td>
<td>INTOCON.INT0IF</td>
</tr>
<tr>
<td>INT1 external interrupt</td>
<td>INTOCON.INT1IP</td>
<td>INTOCON.INT1IE</td>
<td>INTOCON.INT1IF</td>
</tr>
<tr>
<td>TMR0 overflow interrupt</td>
<td>INTOCON.TMR0IP</td>
<td>INTOCON.TMR0IE</td>
<td>INTOCON.TMR0IF</td>
</tr>
<tr>
<td>TMR1 overflow interrupt</td>
<td>INTOCON.TMR1IP</td>
<td>INTOCON.TMR1IE</td>
<td>INTOCON.TMR1IF</td>
</tr>
<tr>
<td>TMR2 overflow interrupt</td>
<td>INTOCON.TMR2IP</td>
<td>INTOCON.TMR2IE</td>
<td>INTOCON.TMR2IF</td>
</tr>
<tr>
<td>CCP1 interrupt</td>
<td>INTOCON.CCP1IP</td>
<td>INTOCON.CCP1IE</td>
<td>INTOCON.CCP1IF</td>
</tr>
<tr>
<td>CCP2 interrupt</td>
<td>INTOCON.CCP2IP</td>
<td>INTOCON.CCP2IE</td>
<td>INTOCON.CCP2IF</td>
</tr>
<tr>
<td>AD converter interrupt</td>
<td>INTOCON.ADIP</td>
<td>INTOCON.ADIE</td>
<td>INTOCON.ADIIF</td>
</tr>
<tr>
<td>USART receive interrupt</td>
<td>INTOCON.RCP</td>
<td>INTOCON.RIE</td>
<td>INTOCON.RIF</td>
</tr>
<tr>
<td>USART transmit interrupt</td>
<td>INTOCON.TCP</td>
<td>INTOCON.TIE</td>
<td>INTOCON.TIF</td>
</tr>
<tr>
<td>Spec. serial port int.</td>
<td>INTOCON.SSP</td>
<td>INTOCON.SSPIE</td>
<td>INTOCON.SSPIIF</td>
</tr>
<tr>
<td>Parallel 14-bit port</td>
<td>INTOCON.P14IP</td>
<td>INTOCON.P14IE</td>
<td>INTOCON.P14IF</td>
</tr>
<tr>
<td>Low-voltage detect int.</td>
<td>INTOCON.LVDP</td>
<td>INTOCON.LVDP</td>
<td>INTOCON.LVDP</td>
</tr>
<tr>
<td>Bus-software interrupt</td>
<td>INTOCON.BSCIP</td>
<td>INTOCON.BSCIE</td>
<td>INTOCON.BSCIF</td>
</tr>
</tbody>
</table>

* SFR can only be used as high-priority interrupt

Example—Initialization to cause Timer1 to generate low priority interrupts

```c
init
    bcf RCON, IPEN ; set PIC for two-level interrupts
    bcf IPR1, TMR1IP ; set timer1 for low priority interrupts
    bcf PIR1, TMR1IF ; clear timer1 overflow flag
    bcf INTOCON, GIEL ; enable low priority interrupts
    bcf INTOCON, GIEH ; enable interrupts
    bcf PIE1, TMR1IE ; enable timer1 interrupts (may want to defer this until later)
```
