CENG-336 Introduction to Embedded Systems Development

Interrupts

Spring 2015

I/O Devices and Modules

- Because of great differences in control methods and transmission modes among various kinds of I/O devices, it is infeasible to connect them directly to the system bus.

- Instead, dedicated I/O modules serve as interfaces between CPU/MPU and peripherals.
  - Controlling device actions, buffering data, error detection, communicating with CPU/MPU

Communication with I/O Modules

- Modern processors provide facilities for being interrupted by external devices.
- This is more efficient than the processor asking devices if they need attention (polling).

- **Programmed I/O** the status of the I/O device is repeatedly checked by the program (polling)

- **Interrupt I/O** the I/O device request the interrupt, and an interrupt handling routine or interrupt service routine is needed

Interrupts

- An **interrupt** is a (temporary) break in the flow of execution of a program
  - the CPU is said to be “interrupted”

- When an interrupt occurs, the CPU deals with the interruption, then carries on where it was left off
  - Program segment dealing with the interrupt is called the "Interrupt Service Routine", ISR.
  - ISR programmer should put the CPU back to its beginning state before returning.

Interrupt Example

1. Suppose you are sitting at home, chatting to someone.
2. Suddenly the telephone rings.
3. You stop chatting, and pick up the telephone to speak to the caller.
4. When you have finished your telephone conversation, you go back to chatting to the person before the telephone rang.

- You can think of:
  - the main routine as you chatting to someone,
  - the telephone ringing causes you to interrupt your chatting (main routine), and
  - the interrupt routine is the process of talking on the telephone.
  - When the telephone conversation has ended, you then go back to your main routine of chatting.

Interrupt Example

- This example is exactly how an interrupt causes a processor to act.
  - The main program is running, performing some function in a circuit,
    - but when an interrupt occurs the main program temporarily halts while another routine (ISR – Interrupt Service Routine) is carried out.
  - When ISR finishes, the processor goes back to the main routine again.

Basic Concepts of Interrupts

- Interrupts are used to overlap computation & input/output tasks
  - Rather than explicitly waiting for I/O to finish, the CPU can attend other tasks
  - Examples would be console I/O, printer output, and disk accesses, all of which are slow processes

- Normally handled by the OS. Rarely coded by programmers under Unix, Linux or Windows.
  - In embedded and real-time systems, however, they are part of the normal programming work.

Basic Concepts of Interrupts

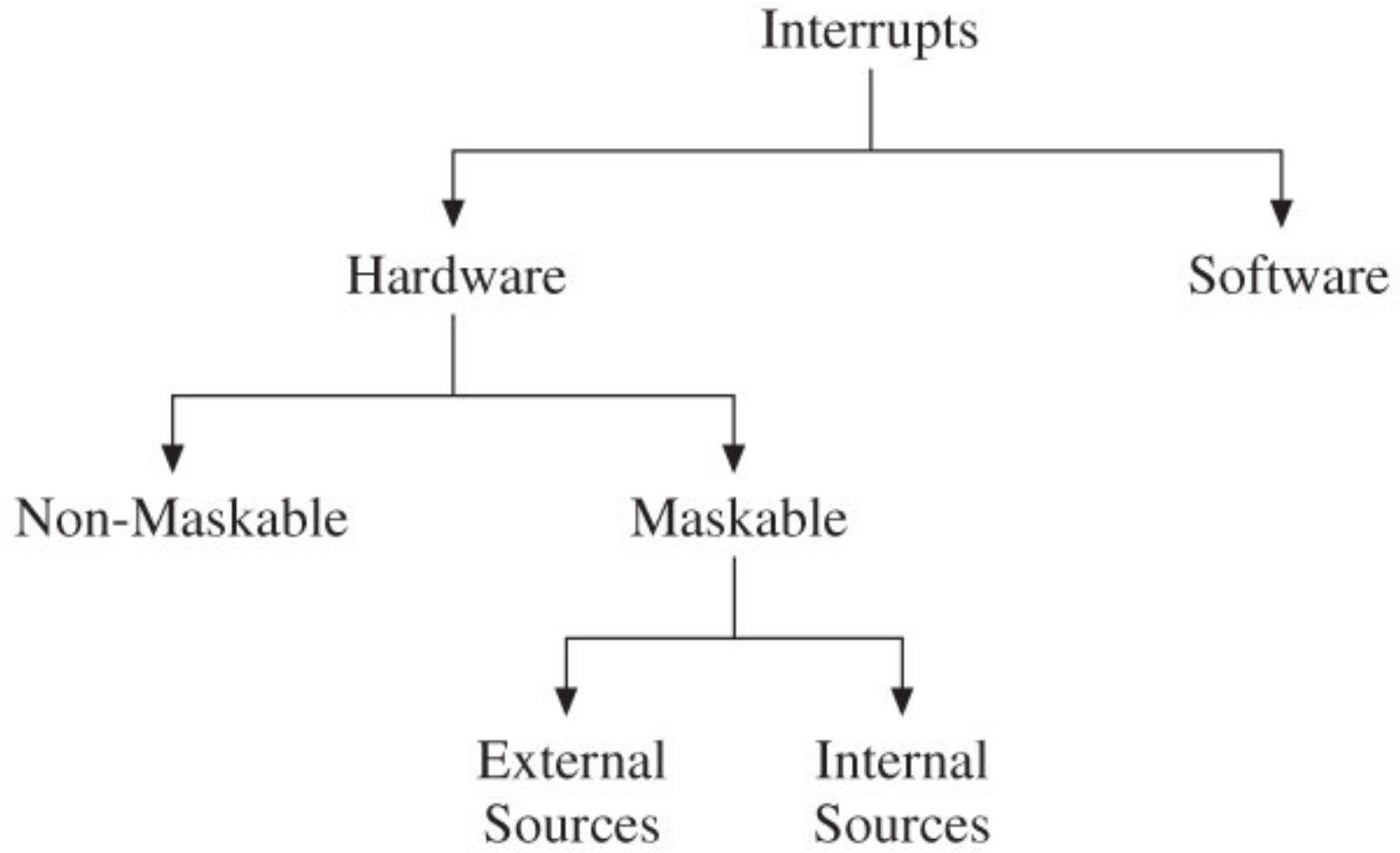
- Why interrupts over polling? Because polling
  - Ties up the CPU in one activity
  - Uses cycles that could be used more effectively
  - Code can't be any faster than the tightest polling loop

- Bottom line: an interrupt is an asynchronous subroutine call (triggered by a hardware event) that saves both the return address and the system status
  - The main difference from a subroutine is that the main routine is unaware that it has been interrupted.

Basic Concepts of Interrupts

- An interrupt is hence a communication process
  - A device
    - Requests the MPU to stop processing
    - Internal or external
  - The MPU
    - Acknowledges the request
    - Attends to the request
    - Goes back to processing where it was interrupted

Types of Interrupts

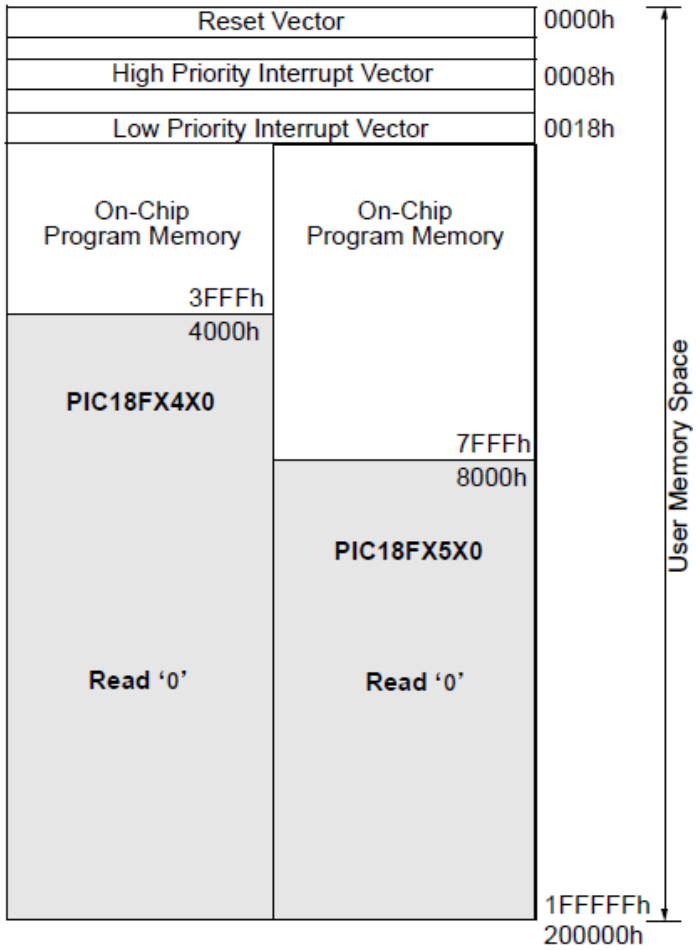


MPU Response to Interrupts

- When interrupts are enabled
  - MPU checks interrupt request flag at the end of each instruction
- If interrupt request is present, the MPU
  - Resets the interrupt flag
  - Saves the return address on the stack
- MPU redirected to appropriate (often fixed) memory location
  - Interrupt vectors
- Interrupt service routine (ISR) meets request
- MPU returns to where it was interrupted
  - Special return instruction needs to be used

The Interrupt Service Routine (ISR)

- When an interrupt is invoked the MPU runs the appropriate Interrupt Service Routine (ISR)
- Interrupt vector table holds the address of ISRs
  - Power-on Reset 0000h
  - High priority interrupt 0008h
  - Low priority interrupt 0018h



Steps in Executing an Interrupt

- Upon noticing the interrupt, the microcontroller
  - Finishes executing the current instruction
  - Pushes the PC of next instruction in the stack
  - Jumps to the interrupt vector table to get the address of ISR and jumps to it
  - Executing ISR instructions until the last instruction of the ISR (RETFIE)
  - Executes the RETFIE instruction
    - Pops the PC from the stack
    - Starts to execute from the address of that PC

PIC18 Interrupts

- The PIC18 Microcontroller family
  - Has multiple sources that can send interrupt requests
  - Does not have any non-maskable or software interrupts
    - All interrupts are maskable hardware
  - Has a priority scheme divided into two groups
    - High priority and low priority
  - Uses many Special Function Registers (SFRs) to implement the interrupt process

PIC18 Interrupt Sources

- External sources
  - Three pins of PORTB
    - RB0/INTO, RB1/INT1,and RB2/INT2
    - Can be used to connect external interrupting sources
      - Keypads or switches
  - PORTB Interrupt (RBI)
    - Change in logic levels of pins RB4-RB7
- Internal peripheral sources
  - Examples
    - Timers
    - A/D Converter
    - Serial I/O
    - ... many others ...

PIC18 Interrupt Sources

- Special Function Registers (SFRs)
  - INTCON
    - Interrupt control and external interrupt sources
  - RCON
    - Priority Enable
  - IPR, PIE, and PIR
    - Internal peripheral interrupts

- Valid interrupt
  - Interrupt request bit (flag)
  - Interrupt enable bit
  - Priority bit

INTCON: Interrupt Control

REGISTER 10-1: INTCON: INTERRUPT CONTROL REGISTER

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(1)
bit 7							bit 0

- Bit 7 of INTCON is called GIE.
  - This is the Global Interrupt Enable (for high priority interrupts).
  - Setting this to 1 tells the PIC that we are going to use interrupts.
- Bit 7 of INTCON is called PEIE/GIEL.
  - Enables peripheral, or low priority interrupts
- Bit 4 of INTCON is called INT0E,
  - INTerrupt 0 Enable. Setting this bit to 1 tells the PIC that RB0 will be an interrupt pin.
- Bit 3 of INTCON is called RBIE,
  - Port B bits 4 to 7 should be used as interrupt sources.

Interrupt Related Flags

- Each interrupt source has three associated bits located in various SFRs
- An "enable" bit to enable/disable the associated interrupt source (e.g. INT0IE)
  - Interrupt requests for disabled sources are ignored by the microcontroller
- A "status" bit, also called interrupt flag to indicate that an interrupt request for the associated source was received (e.g. INT0IF)
  - ISR uses these flags to identify which source generated the interrupt.
  - Must be cleared by the programmer before exiting ISR
- A priority bit to select high (1) or low (0) priority

RCON: Reset Control

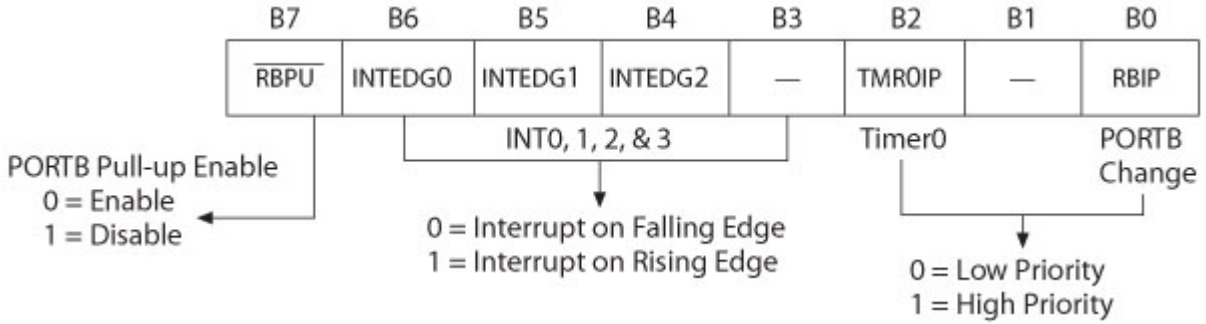
REGISTER 4-1: RCON: RESET CONTROL REGISTER

R/W-0	R/W-1(1)	U-0	R/W-1	R-1	R-1	R/W-0(2)	R/W-0
IPEN	SBOREN	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

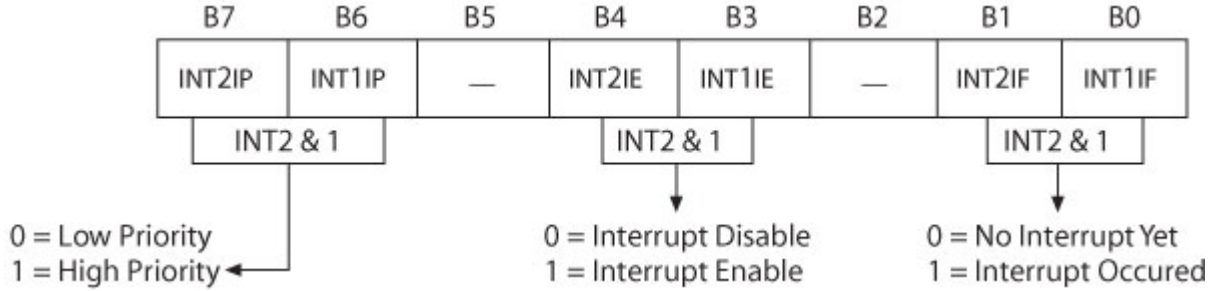
- Bit 7, IPEN, enables (1) interrupt priorities
  - Interrupt priorities
    - High-priority interrupt vector 000008_H
    - Low-priority interrupt vector 000018_H
    - A high-priority interrupt can interrupt a low-priority interrupt in progress.
  - When disabled, only high priority in 0008h is used (compatibility)
- Bits 4-0 are flags to indicate reset source

INTCON2, INTCON3

- INTCON2 Register



- INTCON3 Register



External interrupt on RB0/INT0

- External interrupt on RB0/INT0 pin is triggered by
  - rising edge, if bit INTEDG0=1 (in INTCON2 register),
  - falling edge, if INTEDG0=0.
- When correct signal appears on INT0 pin, INT0F bit is set in INTCON register.

- INTF bit (INTCON<1>) must be cleared (reset) in interrupt routine, so that interrupt would not occur again while going back to the main program.
  - This is an important part of the interrupt programming, which the programmer must not forget. Otherwise, program will constantly go into interrupt routine.
- Interrupt can be turned off by resetting INT0E control bit.

Interrupt [Pins 4-7 of Port B]

- Change of input signal on PORTB <7:4> sets the RBIF bit.
- Four pins RB7 to RB4 of port B, can trigger an interrupt which occurs when status on them changes from logic one to logic zero, or vice versa *since the last read*.

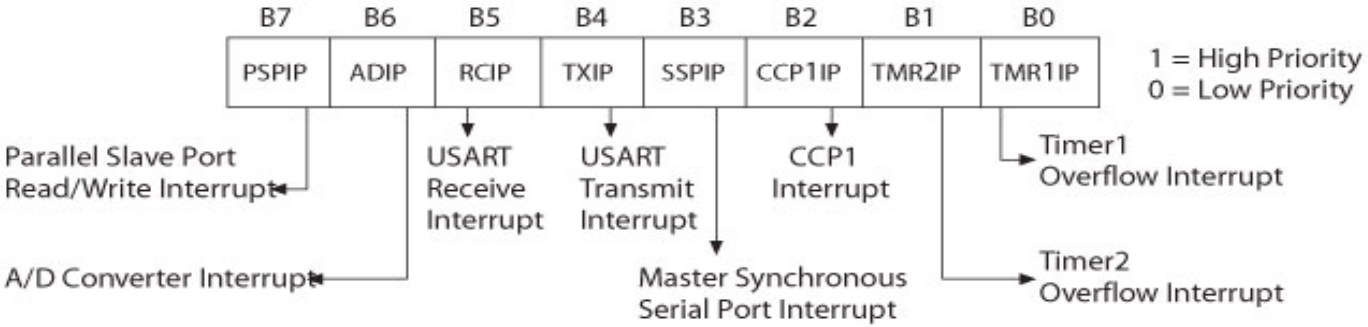
- For pins to be sensitive to this change, they must be defined as inputs.
  - If any one of them is defined as output, interrupt will not be generated at the change of status.
  - If they are defined as input, their current state is compared to the old value which was stored at the last reading from port B.
- Interrupt can be turned on/off by setting/resetting the RBIE bit in the INTCON register.

Internal Interrupts

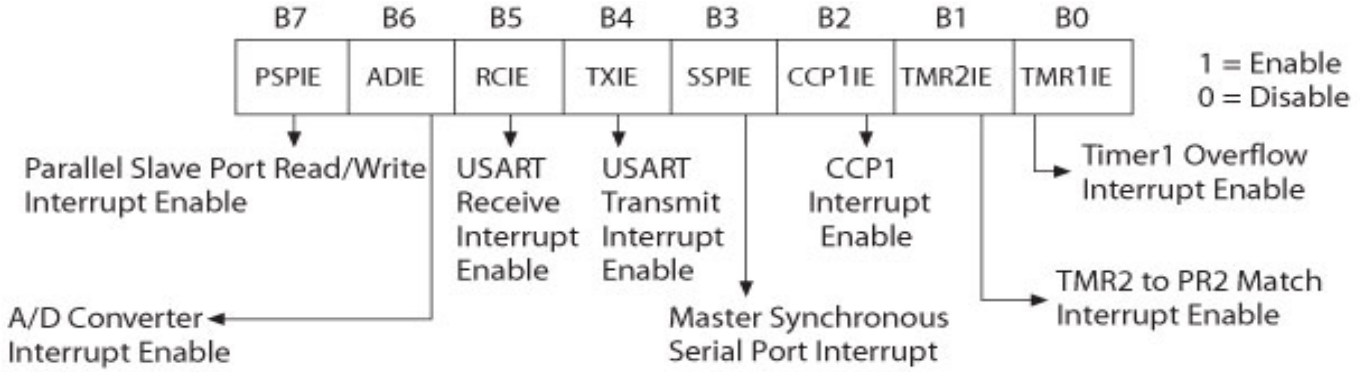
- PIC18 MCU internal interrupt sources
  - Timers
  - A/D converter
  - Serial I/O
- Each interrupt has three bits
  - Interrupt priority bit
  - Interrupt enable bit
  - Interrupt request bit (flag)
- Interrupt registers
  - IPR: Interrupt Priority Register
  - PIE: Peripheral Interrupt Enable
  - PIR: Peripheral Interrupt Request (Flags)

Internal Interrupt Registers

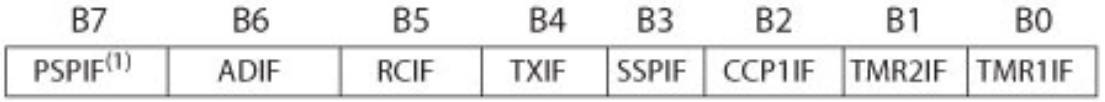
- IPR1



- PIE1



- PIR1



Interrupt Initialization

- In order to use an interrupt mechanism of a microcontroller, some preparatory tasks need to be performed (a.k.a "initialization“).
- During initialization we must define
  - to what interrupts the microcontroller will respond
  - and which ones it will ignore.
- If we do not set the enable bit that allows a certain interrupt, program will not execute an interrupt subprogram.
- Through this we can control how to respond to interrupts

```
clrf INTCON          ; All interrupts disabled to begin with
movlw B'00010000'
movwf INTCON         ; This enables the external interrupt only
bsf INTCON, GIE      ; Finally, enable all interrupts
```

Interrupt Service Routine (ISR)

- Similar to a subroutine
- Attends to the request of an interrupting source
  - Clears the interrupt flag
  - Should save register contents that may be affected by the code in the ISR
  - Must be terminated with the instruction RETFIE
- When an interrupt occurs, the MPU:
  - Completes the instruction being executed
  - Disables global interrupt enable
  - Places the return address on the stack

Interrupt Service Routine (ISR)

- High-priority interrupts
  - The contents of W, STATUS, and BSR registers are automatically saved into respective shadow registers.

- Low-priority interrupts
  - These registers **must be saved** as a part of the ISR by the programmer if they are to be used

- RETFIE [s] ;Return from interrupt
- RETFIE FAST ;FAST equivalent to s = 1
  - If s =1: MPU also retrieves the contents of W, BSR, and STATUS registers

Multiple Interrupt Sources

- All interrupt requests are directed to one of two memory locations (interrupt vectors)
  - 000008_H (high-priority)
  - 000018_H (low-priority)

- When there are multiple requests
  - The interrupt source must be identified by checking the interrupt flags

Program Organization in MPLAB

- When you first power up the PIC, or if there is a reset,
  - Program Counter points to address 0000h, which is right at the start of the program memory.
- Any time there is an interrupt from a high priority source, the Program Counter will point to address 0008h.
- Any time there is an interrupt from a low priority source, the Program Counter will point to address 0018h.
- When we are writing our programs
  - First, we have to tell the PIC to jump over addresses 0008h and 0018h for the main routine
  - Place jump instructions on addresses 0008h and 0018h to high and low priority interrupt service routines, keeping them separate from the rest of the program.

Program Organization in MPLAB

- Start the program with a command called ORG.
  - This command means Origin, or start.
  - We follow it with an address.
  - Because the PIC will start at address 0000h, we type "ORG 0000h".
- Next we need to skip over 0008h and 0018h.
  - We do this by placing a GOTO instruction, followed by a label which points to our main program.

```
ORG    0000h   ;PIC starts here on power up and reset
GOTO  start    ;Goto our main program
```

Program Organization in MPLAB

- We then follow this GOTO command two additional ORG directives
  - These define where our interrupt service routines are located
  - We use a GOTO statement to the high priority interrupt service routine.
  - We can then simply write the low priority ISR routine at 0018h

```
ORG   0008h     ; PIC jumps here on high priority int.
GOTO  high_isr  ; Goto the high priority ISR
ORG   0018h     ; PIC jumps here on low priority int.
...             ; The low priority ISR goes here
RETFIE
```

Important Notes

- Checking interrupt sources
  - All interrupts jump to either the low or high priority vector
  - Your ISR must check interrupt flag bits to determine which source generated the interrupt
- Clearing interrupt flags
  - Once you detect a particular source (flag=1), you must clear it before you return from the ISR
  - Otherwise, the same interrupt will be triggered again, resulting in the ISR being repeatedly called
  - This is called "acknowledging the interrupt"
- Saving context (W and STATUS)
  - Ensures that interrupted code can continue execution
  - Must be done explicitly for the low priority ISR

Program Organization in MPLAB

```
        ORG 0x0000
        goto    Main          ;go to start of main code

;*****************************************************************
;High priority interrupt vector
        ORG 0x0008
        bra HighInt     ;go to high priority interrupt routine

;*****************************************************************
;Low priority interrupt vector and routine

        ORG 0x0018
;   *** low priority interrupt code goes here ***
        retfie

;*****************************************************************
;High priority interrupt routine
HighInt:
;   *** high priority interrupt code goes here ***
        retfie  FAST
;*****************************************************************
;Start of main program
; The main program code is placed here.
Main:
;   *** main code goes here ***

        END
```

Illustration

- Problem Statement
  - INT1 set up as a high-priority interrupt
  - Timer1 and Timer2 set up as low-priority
  - Identify the interrupt sources
    - Execute the appropriate interrupt service routines

Illustration

Label	Opcode	Operand	Comments
S_TMP	EQU	0x100	;Temp Registers
W_TMP	EQU	0x101	
	ORG	0x00	
	GOTO	MAIN	
	ORG	0x08	;High-Priority Interrupt Vector
INTCHK:	GOTO	INT1_ISR	
	ORG	0x18	;Low-Priority Interrupt Vector
TIMERCHK:	BTFSC	PIR1,TMR1IF	;Timer1 Flag, Skip if Clear
	GOTO	TMR1_ISR	
	BTFSC	PIR1,TMR2IF	;Timer2 Flag, Skip if Clear
	GOTO	TMR2_ISR	
	RETFIE		

Illustration

Label	Opcode	Operand	Comments
MAIN:			;Main Program goes here
			;Do Something
HERE:	GOTO	HERE	;Wait for an Interrupt
	ORG	0x100	
INT1_ISR:	BCF	INTCON3,INT1IF	;Clear Flag
			;Do Something
	RETFIE	FAST	;Retrieve registers and Return

Illustration

Label	Opcode	Operand	Comments
TMR1_ISR:	MOVFF	STATUS, S_TMP	
	MOVWF	W_TMP	;Save Registers
	BCF	PIR1,TMR1IF	;Clear Flag
			;Do Something
	MOVF	W_TMP,W	;Retrieve Registers
	MOVFF	S_TMP,STATUS	
	RETFIE		;Return from interrupt
TMR2_ISR			;Similar to Timer1

Programming with Interrupts

Extending the Round-Robin Loop

- Round-Robin with Interrupts
  - Keeps the same infinite loop structures
  - Uses interrupts to respond to "asynchronous" events quickly, informing tasks in the main loop

```
void interrupt high_priority high_isr() { ... }
void interrupt low_priority low_isr() { ... }

main() {
      initialize();
      while (1) {
              task_1();
              ...
              task_n();
      }
}
```

Round-Robin with Interrupts

- Before, each task was explicitly polling for events
  - If there are too many tasks, a short event might be detected late, or even missed
  - Multiple, successive occurrences of the same event might go undetected

- Now, the ISR immediately responds to the event
  - Performs all time-critical tasks to do
  - Alerts the main routine through a flag or buffers for less time-critical tasks

Simple Example with Interrupts

- Same as before: Use a button to toggle an LED
- Previous design:
  - button_task monitors the button on RB0
  - led_task controls the LED on RB1
- New design:
  - RB0 is an external interrupt source! We can detect the rising edge with an interrupt!
  - So, replace the button_task with an ISR
  - Keep the LED task as before

Simple Example with Interrupts

```
unsigned char toggle_flag;

void init_interrupts() {
    /* Disable priorities */
    RCONbits.IPEN = 0;
    /* Enable INT0 source */
    INTCON = 0x10;

    /* Once all configuration is done,
       enable all interrupts */
    INTCONbits.GIE = 1;
}
void init_ports() {
    TRISB = 0xFD;
}
void main() {
    toggle_flag = 0;
    init_ports();
    init_interrupts();
    while(1) {
            led_task();
    }
}
```

```
void led_task() {

    if (toggle_flag != 0) {
            PORTB = PORTB ^ 0x02;
            toggle_flag = 0;
    }

}

void interrupt high_priority high_isr() {

    /* Check flags to detect which source
       triggered the interrupt
    if (INTCONbits.INT0IF) {
            /* Clear interrupt flag. */
            INTCONbits.INT0IF = 0;

            /* Inform LED task */
            toggle_flag = 1;
    }

}
```

More Complex Example

- Monitor a digital input on RB0, when it is pressed:
  - read the 8 bit value from PORTC
  - send it serially, one bit at a time from PORTD bit 0
  - ensuring that there is at least 50ms between each bit.
- Simple possible solution:

```
main() {
    unsigned char value, count;
    initialize();
    while (1) {
        if (PORTB & 0x01 != 0) {
            value = PORTC;
            for (count = 0; count < 8; count++) {
                if (value & (0x01 << count))        PORTD |= 0x01;
                else                                PORTD &= 0xfe;
                busy_wait(50);
    } } }
}
```

Problems with the Simple Solution

- Continuously reading RB0 is problematic, possibly triggering multiple reads of PORTC on one button press
  - That's easy, use state machines just like before
- What if RB0 is pressed again while we were sending data on PORTD?
  - That's why we need to avoid busy waits. They prevent other parts of the code from running
  - Also, if RB0 is pressed multiple times before we have a chance to send all data, we need to buffer data!
- In any case, we better detect that RB0 is pressed quickly and buffer the reading of PORTC

Studying the Problem

- Event sources:
  - RB0 is pressed (INT0 interrupt can be used)
  - Sending of the current bit completed (i.e. 50ms elapsed)
  - Sending of the current byte completed (all 8 bits done)
- An alternative design
  - INT0 interrupt service routine reads PORTB and places the data in a buffer (FIFO, ring buffer)
  - send_task() monitors the buffer, starts sending when a new byte arrives. We should use state machines for this.
  - When a bit is written to PORTD, send_task() holds off for 50ms until the next bit.
- This is a typical use of round-robin with interrupts

Implementation

- Initialization and main

```
void init_interrupts() {
    RCONbits.IPEN = 0;          /* Disable priorities */
    INTCON = 0x10;              /* Enable external interrupts */
}
void init_ports() {
    TRISB = 0xFF;               /* PORTB is all inputs */
    TRISC = 0xFF;               /* PORTC is all inputs */
    TRISD = 0xFE;               /* PORTD pin0 is an output */
}
main() {
    init_ports();
    init_interrupts();
    INTCONbits.GIE = 1;         /* Enable all interrupts

    while (1) {
       send_task();
    }
}
```

Ring buffers

- Useful for buffering incoming/outgoing data

```
#define BUFSIZE 16                  /* Static buffer size. Maximum amount of data */
unsigned char buffer[BUFSIZE];              /* No malloc's in embedded code! */
unsigned char head = 0, tail = 0;           /* head for pushing, tail for popping */

bool buf_isempty() { return (head == tail); }

void buf_push( unsigned char v ) {          /* Place new data in buffer */
    buffer[head++] = v;
    if (head == tail) { /* Overflow!!! */ error(); } else {
        if (head == BUFSIZE) head = 0;
    }
}

unsigned char buf_pop() {                   /* Retrieve data from buffer */
    unsigned char v;
    if (buf_isempty()) { /* Underflow!! */ error(); } else {
        v = buffer[tail++];
        if (tail == BUFSIZE) tail = 0;
    }
}
```

INT0 Interrupt Service Routine

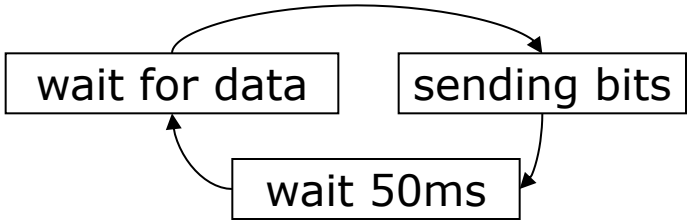
- Main function is to read PORTB and buffer data

```
void interrupt high_priority high_isr() {
    unsigned char v;

    if (INTCONbits.INT0IF) {                    /* Check flags to make identify source */

        INTCONbits.INT0IF = 0;                  /* Clear interrupt flag */

        v = PORTB;                              /* Read data */
        buf_push( v );                  /* Push into buffer */
    }
}                                               /* Done! */
```

- The ISR is short! It performs only the most urgent task, which is reading and storing data

Task for sending data

- Once again, use state machines




```
unsigned char send_state = 0;
void send_task() {
    unsigned char sending, send_count;
    switch (send_state) {
    case 0:                         /* Wait for new data */
            disable_int0();                     /* IMPORTANT: Prevents corruption of buffer! */
            if (!buf_isempty()) {
                sending = buf_pop();
                send_state = 1; send_count = 0; }
            enable_int0();                      /* Re-enable interrupts to resume operation */
            break;
    case 1:                                     /* Send next bit */
            if ( (sending & (0x01 << send_count) != 0)       PORTD = PORTD | 0x01;
            else                                            PORTD = PORTD & ~0x01;
            send_count++;   send_state = 2;
            break;
    case 2:                                     /* Skip until 50ms elapses */
            if (time_since_laststate() > 50) send_state = 0;
            break;
    }
}
```

Observations

- No busy waits at all!
- Button press on RB0 immediately results in reading of data on PORTC
- 50ms waiting is accomplished by send_task() staying in state 2 until 50ms elapses since last state transition
  - we can use timers for this, stay tuned for the next lecture
- Ring buffers are extremely useful for communicating between ISRs and tasks
- Data Sharing: Disabling/enabling interrupts in send_task() to prevent buffer corruption