The University of Texas at Arlington

Lecture 11
Interrupts

CSE@UTA

CSE 3442/5442

Embedded Systems 1

Based heavily on slides by Dr. Gergely Záruba and Dr. Roger Walker

External PIC Influence

- We don’t always just want data transfer from pin connections (input or output)
  - Regardless of the data’s value, the same actions are performed
- Instead we can have the values on pin connections influence or control which segments of code/functions are used
  - Based off input, we can handle things in a different particular manner
- ***An <u>Interrupt</u> uses hardware to cause special software execution***

Polling vs. Interrupts

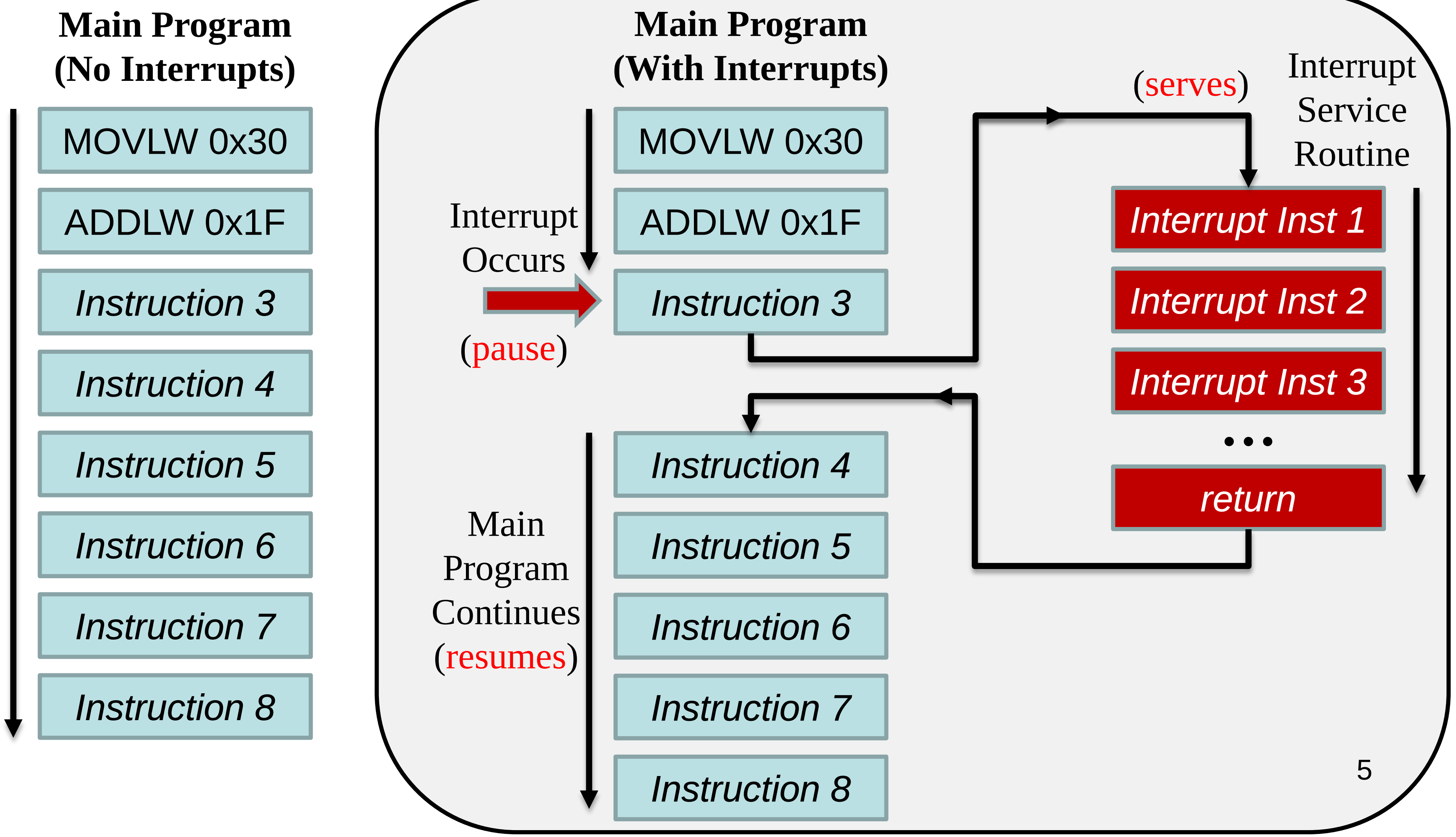
- **Polling**
  - Continuously monitor the status of a device, bit, or pin
  - When the condition is met, perform the service
  - Wastes the PIC's time and resources
    - Only "looking" at a single location
  - Can get stuck (infinite loop) if condition is never met
  - Could miss other important input data or events

```
            while(1)
            {
                  if(PORTBbits.RB0 == 1)
                        break;
            }
      //or
            while(PORTBbits.RB0 == 0);
```

Polling vs. Interrupts

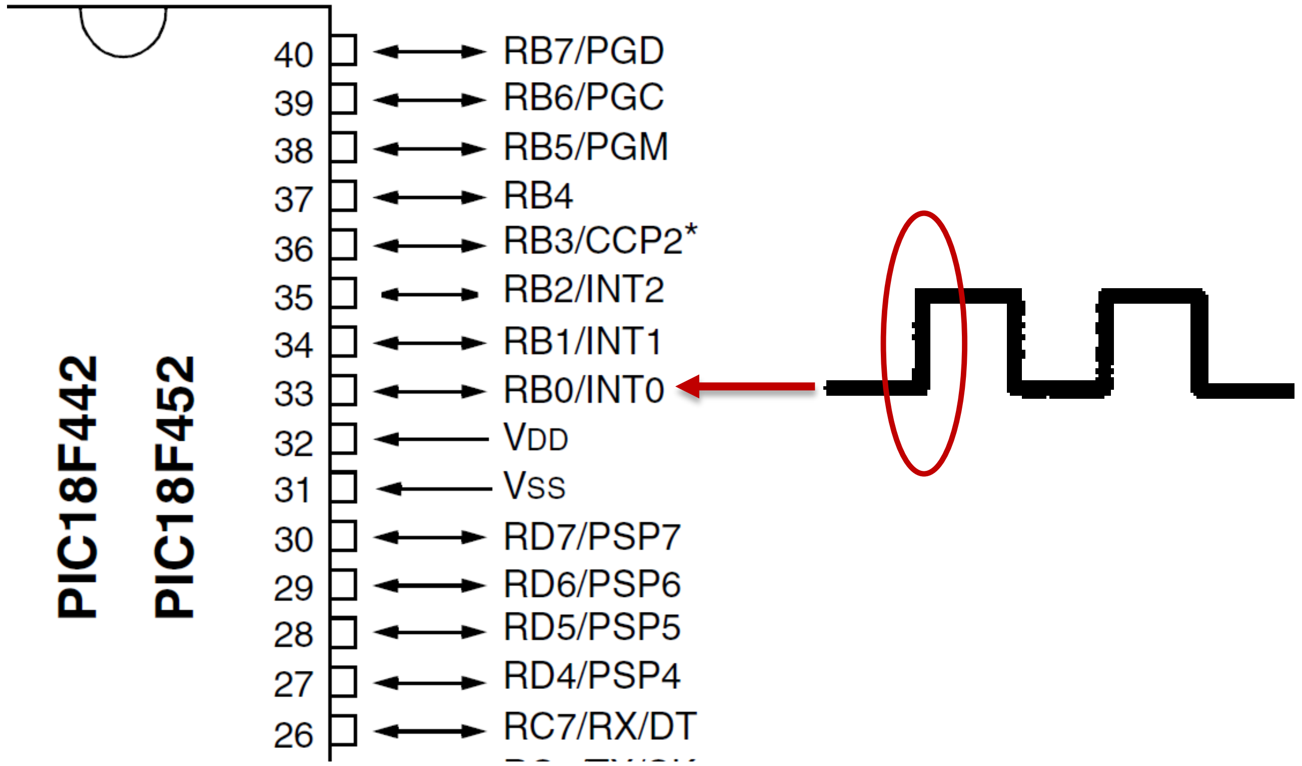
- **Interrupts**
  - Whenever a device (pin, peripheral) needs the PIC's service, it notifies by sending an interrupt signal
    - Asynchronous (can happen at any time)
  - When that signal is detected…
    - PIC stops (**pauses**) its current actions
    - Handles (**serves**) the source of the interrupt
    - Returns exactly where the PIC left off (**resumes**)
  - Doesn't bog-down the PIC's resources
  - Can serve many devices (multiple interrupt sources)
    - Each can get the PIC's attention at any time
  - Can assign priorities to each interrupt
    - "Interrupt an interrupt"
  - Can also ignore (mask) interrupt sources at any time
  - When sleeping, they can wake up the microcontroller

Interrupt Handling



Basic Example

- When PORTB pin B0 is brought HIGH (1)
  - Go to function → ***Input_Detected()***



Basic Example (No Interrupts)

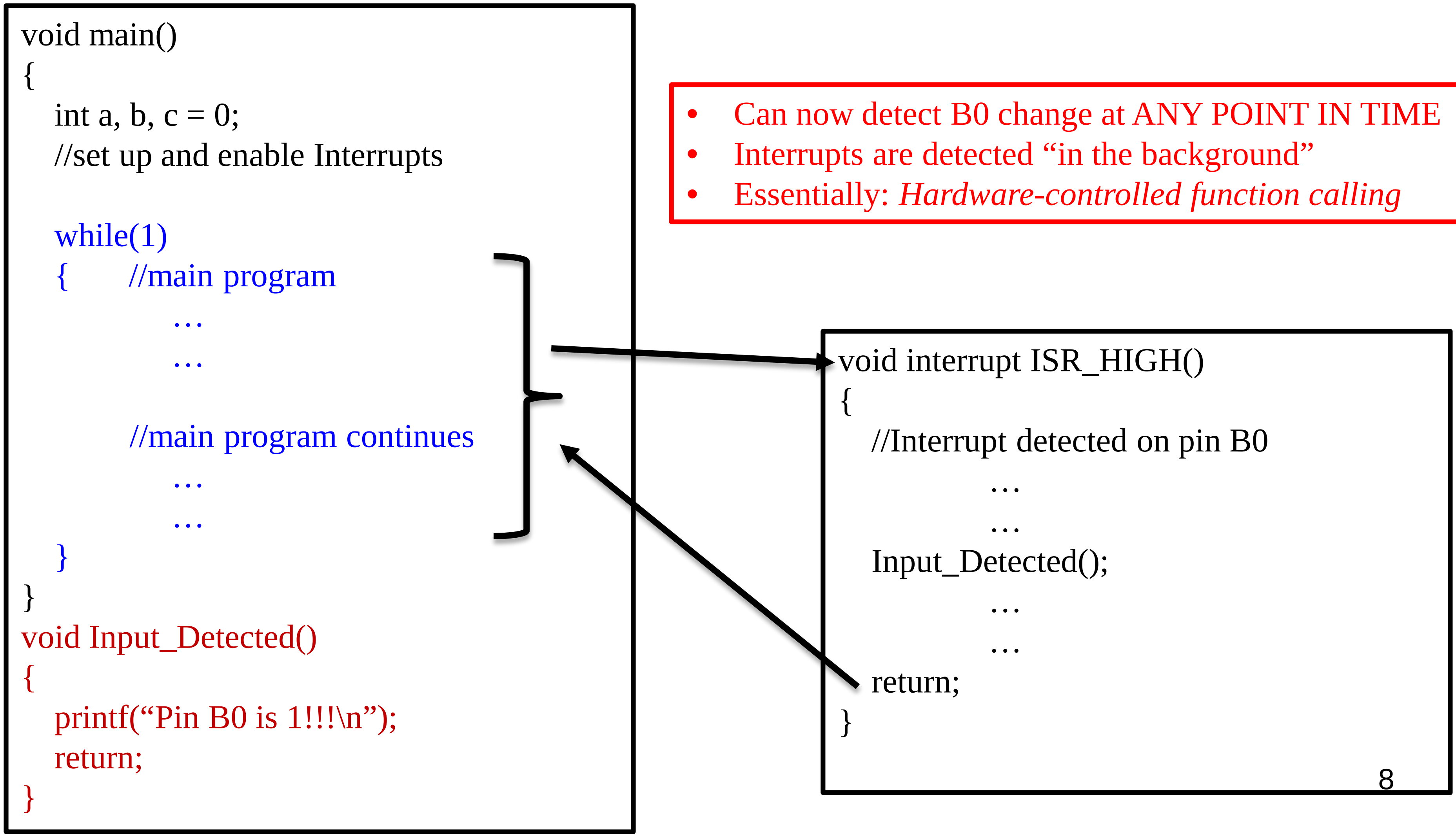
```
void main()
{
    int a, b, c = 0;
    while(1)
    {       //main program
                …

            if(PORTBbits.B0 == 1)
                Input_Detected();

            //main program continues
                …
                …
    }
}
void Input_Detected()
{
    printf("Pin B0 is 1!!!\n");
    return;
}
```

OR

```
void main()
{
    int a, b, c = 0;
    while(1)
    {       //main program
                …

            while(PORTBbits.B0 == 0);
                        //wait here until B0 is 1
            Input_Detected();

            //main program continues
                …
    }
}
void Input_Detected()
{
    printf("Pin B0 is 1!!!\n");
    return;
}
```

Basic Example (With an Interrupt)

- Can now detect B0 change at ANY POINT IN TIME
- Interrupts are detected “in the background”
- Essentially: *Hardware-controlled function calling*



```
void main()
{
   int a, b, c = 0;
   //set up and enable Interrupts

   while(1)
   {       //main program
                …
                …

           //main program continues
                …
                …
   }
}
void Input_Detected()
{
   printf(“Pin B0 is 1!!!\n”);
   return;
}
```

```
void interrupt ISR_HIGH()
{
   //Interrupt detected on pin B0
                …
                …
   Input_Detected();
                …
                …
   return;
}
```

Interrupt Service Routines

- Where to “jump” when an interrupt is triggered?
  - Need a sub-routine to handle interrupts
- **Interrupt Service Routines** (ISR) serve that purpose
- The ISRs have a fixed location in Program ROM
  - If multiple ISRs, the group of locations is the **interrupt vector table**
- PIC18 only has **three** locations to handle interrupts but we only have control of **two** of them for “normal” interrupts
  - **Program ROM: 0x0008 – HIGH Priority**
  - **Program ROM: 0x0018 – LOW Priority**

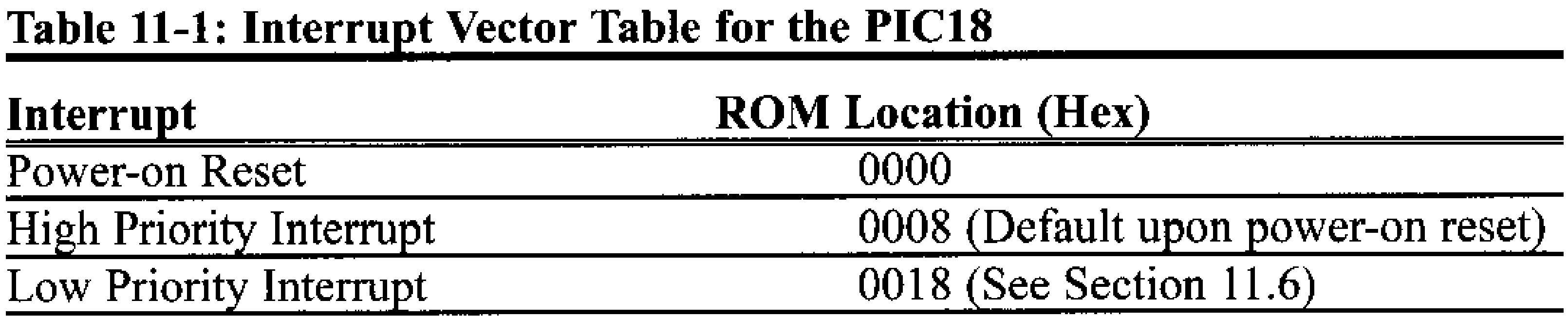
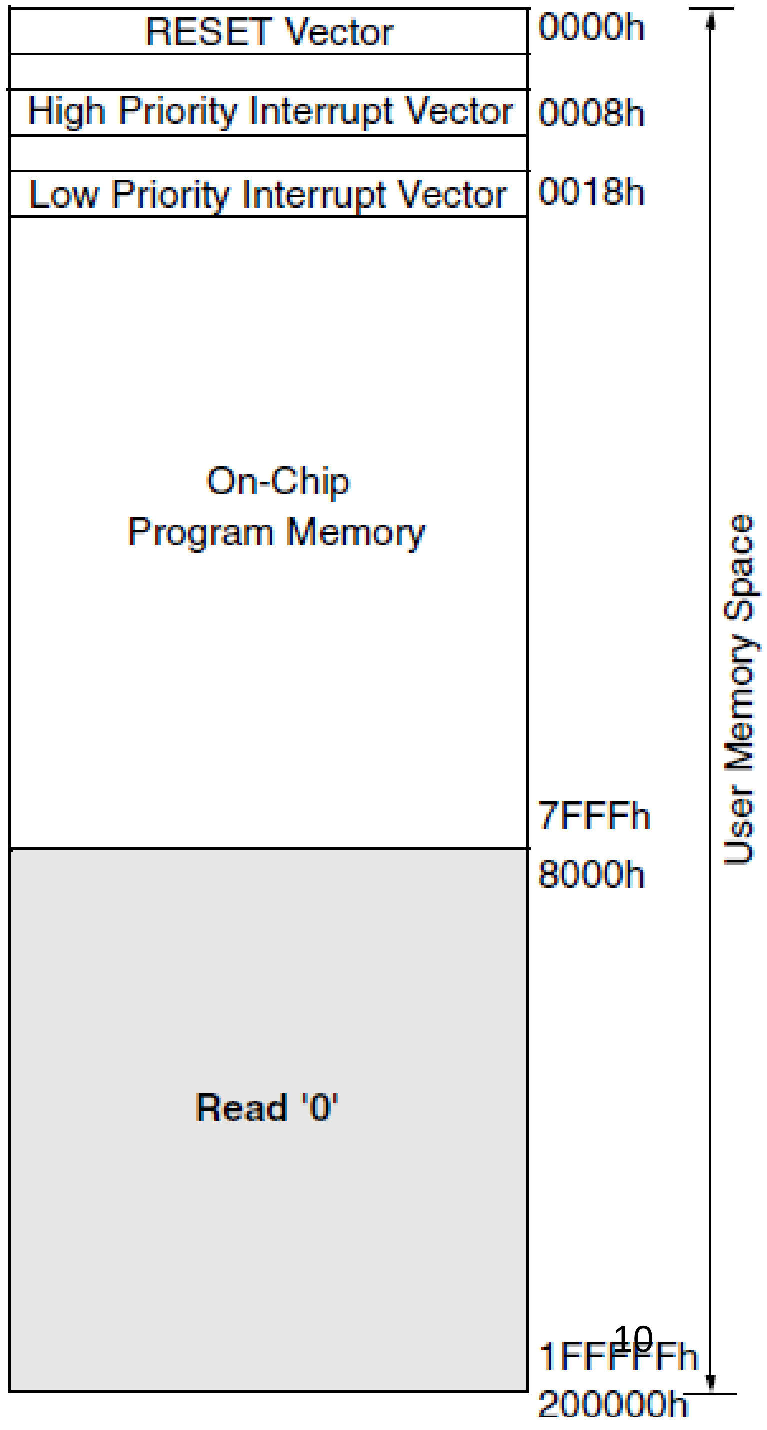


Table 11-1: Interrupt Vector Table for the PIC18

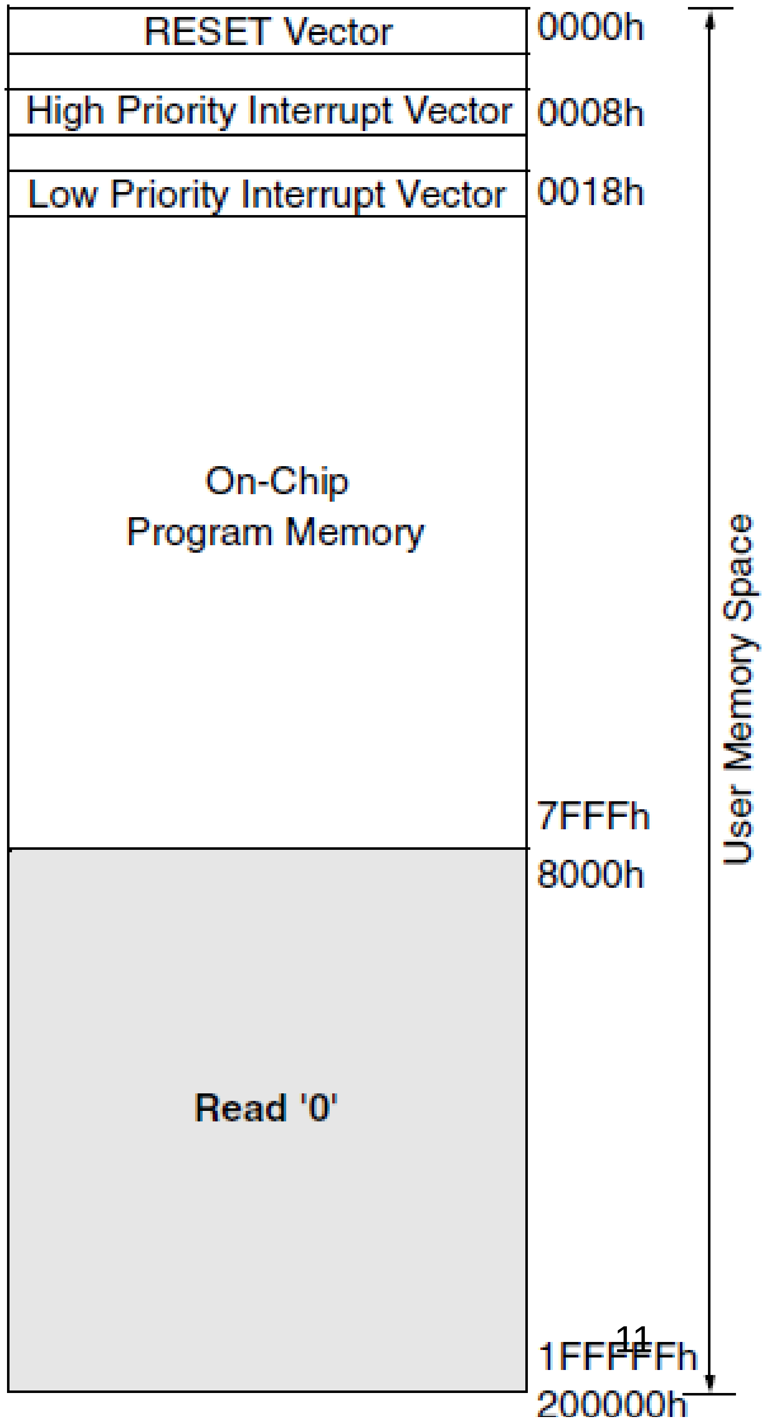
Interrupt	**ROM Location (Hex)**
Power-on Reset	0000
High Priority Interrupt	0008 (Default upon power-on reset)
Low Priority Interrupt	0018 (See Section 11.6)

As there is limited space at these addresses it is a good idea to place a **GOTO** instruction at the interrupt vector jumping to a remote location



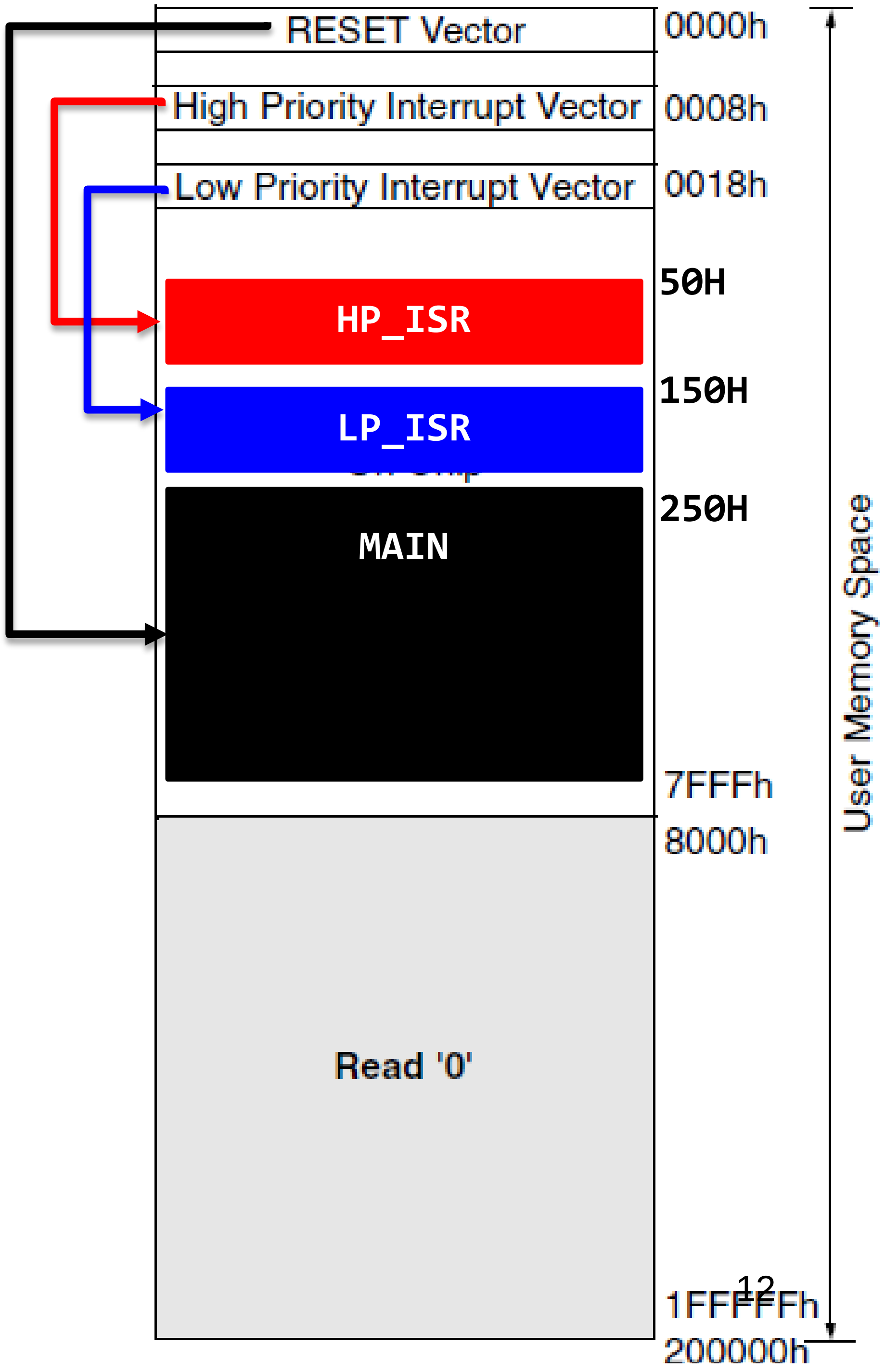
As there is limited space at these addresses it is a good idea to place a **GOTO** instruction at the interrupt vector jumping to a remote location

```
            ORG     0000H
            GOTO    MAIN

            ORG     0008H
            GOTO    HP_ISR

            ORG     0018H
            GOTO    LP_ISR


            ORG     50H
HP_ISR      …

            ORG     150H
LP_ISR      …

            ORG     250H
MAIN        …
```



As there is limited space at these addresses it is a good idea to place a **GOTO** instruction at the interrupt vector jumping to a remote location

```
        ORG     0000H
        GOTO    MAIN

        ORG     0008H
        GOTO    HP_ISR

        ORG     0018H
        GOTO    LP_ISR


        ORG     50H
HP_ISR  …

        ORG     150H
LP_ISR  …

        ORG     250H
MAIN    …
```



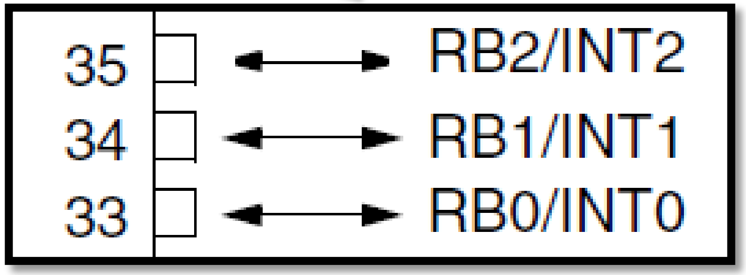
CSE@UTA

What Happens When an Interrupt Hits?

1. The current instruction's execution is finished and the next instruction's address is pushed to the stack
   - Interrupts are disabled for HP (GIE or GIEH or GIEL cleared)
2. The PC is loaded with the interrupt vector
   - Jump to the ISR
3. The instructions in the ISR are executed until a **RETFIE** instruction
   - Return From Interrupt Exit
4. RETFIE will cause the microcontroller to pop the PC from the stack and resume normal operations
   - Interrupts are re-enabled (GIE or GIEH or GIEL set)

Sources of Interrupts

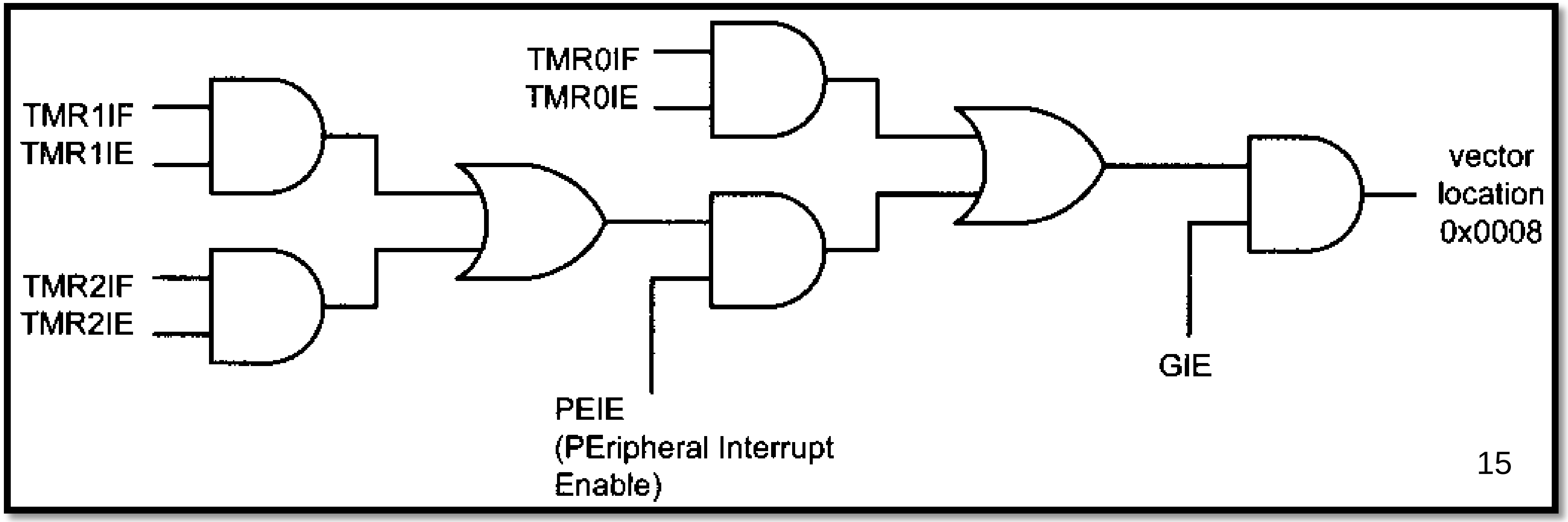
1. Timers
2. Hardware Interrupts (external pins, INT)
   - PORTB: RB0, RB1, and RB2
3. Serial Communication
   - Receive and Transmit
4. PORTB-Change
5. ADC
6. CCP/PWM



CSE@UTA

Simplified View of Interrupts

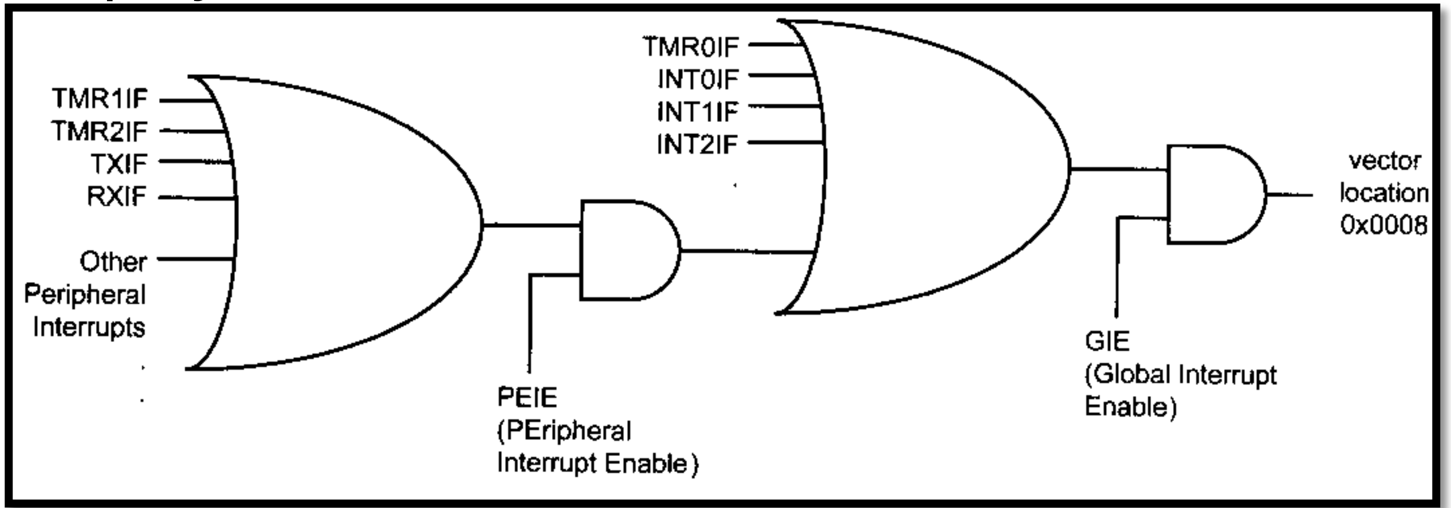
- We can easily think of an interrupt as **two** digital signals:
  1. **Enable bit** can allow/disallow the actual interrupt from happening (Enabled = Unmasked, Disabled = Masked)
  2. **Flag bit** is set if interrupt **should be** invoked (something happened)



CSE@UTA

Simplified View of Interrupts

- We can easily think of an interrupt as **two** digital signals:
  1. **Enable bit** can allow/disallow the actual interrupt from happening (Enabled = Unmasked, Disabled = Masked)
  2. **Flag bit** is set if interrupt **should be** invoked (something happened)
- To make things confusing, some peripherals can be masked in a group by a PEIE mask


TMR1IF
TMR2IF
TXIF
RXIF
Other Peripheral Interrupts
PEIE (PEripheral Interrupt Enable)
TMR0IF
INT0IF
INT1IF
INT2IF
GIE (Global Interrupt Enable)
vector location 0x0008

CSE@UTA

Masking Interrupts

- By default, all interrupts are **masked** (disabled)
  - The PIC will not respond to any interrupts
- Up to the user to enable them if they are needed
- Enabling/disabling interrupts is done through designated registers in the SFR:
  - INTCON, INTCON2, INTCON3
  - RCON
  - PIR1, PIR2
  - PIE1, PIE2
  - IPR1, IPR2
- All interrupts can be masked by clearing the **GIE** (Global Interrupt Enable) bit in **INTCON** (default)

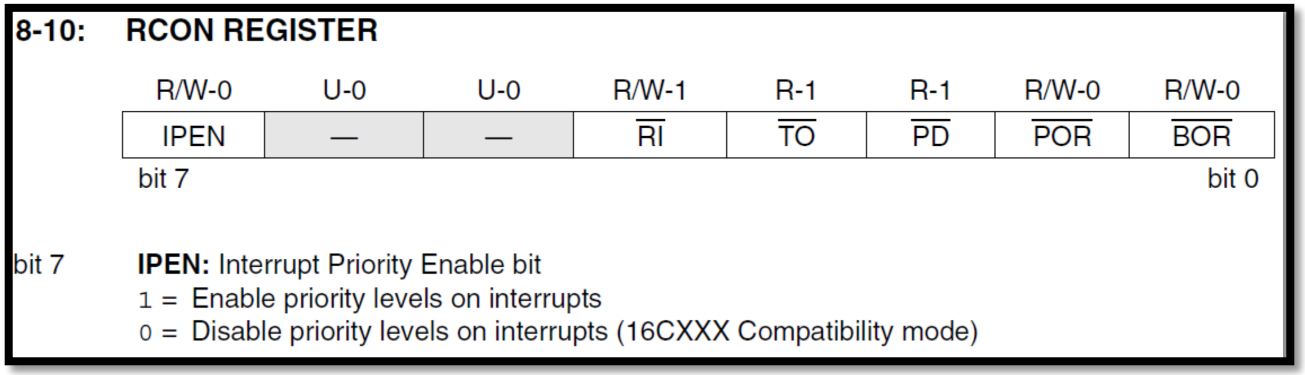
CSE@UTA

Enabling Interrupts

1. Allow the specific interrupt to occur
   - INTCONbits.TMR0IE = 1 (timer 0 can interrupt)
   - INTCON3bits.INT1IE = 1 (INT 1 can interrupt)
   - PIE1bits.ADIE = 1 (ADC can interrupt)
   - …
2. If specific interrupt falls into Peripheral category, must also enable another bit
   - INTCONbits.PEIE = 1
3. Allow any interrupt to occur
   - INTCONbits.GIE = 1

Two Levels of Priorities RCONbits.IPEN

- The PIC18 has two levels of interrupts: **HIGH** and **LOW**
- By default (when reset) all interrupts are high priority (00008H)
- In **RCON** we can enable the two-level priority option



8-10: RCON REGISTER

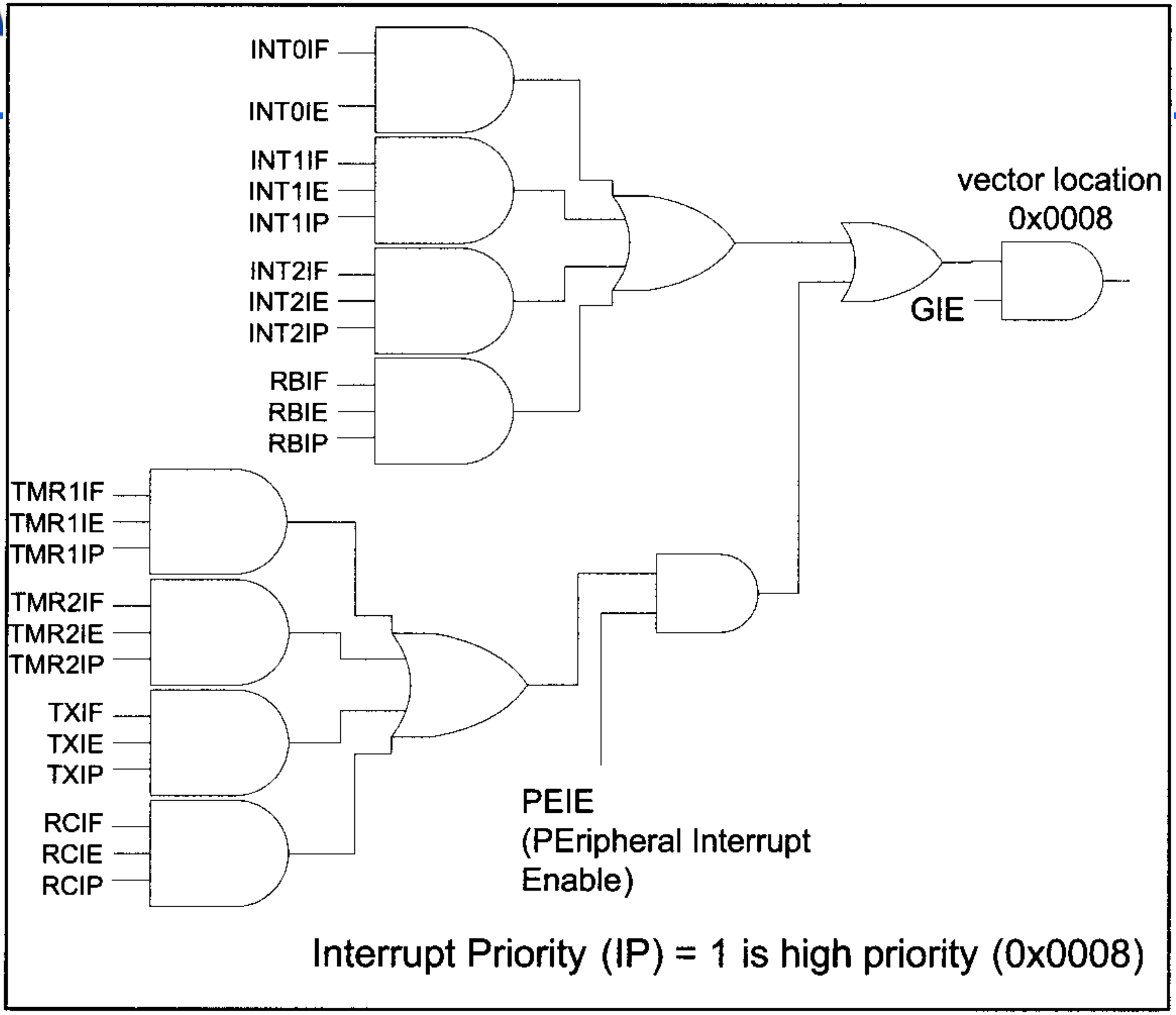
R/W-0	U-0	U-0	R/W-1	R-1	R-1	R/W-0	R/W-0
IPEN	—	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

bit 7 **IPEN:** Interrupt Priority Enable bit
1 = Enable priority levels on interrupts
0 = Disable priority levels on interrupts (16CXXX Compatibility mode)

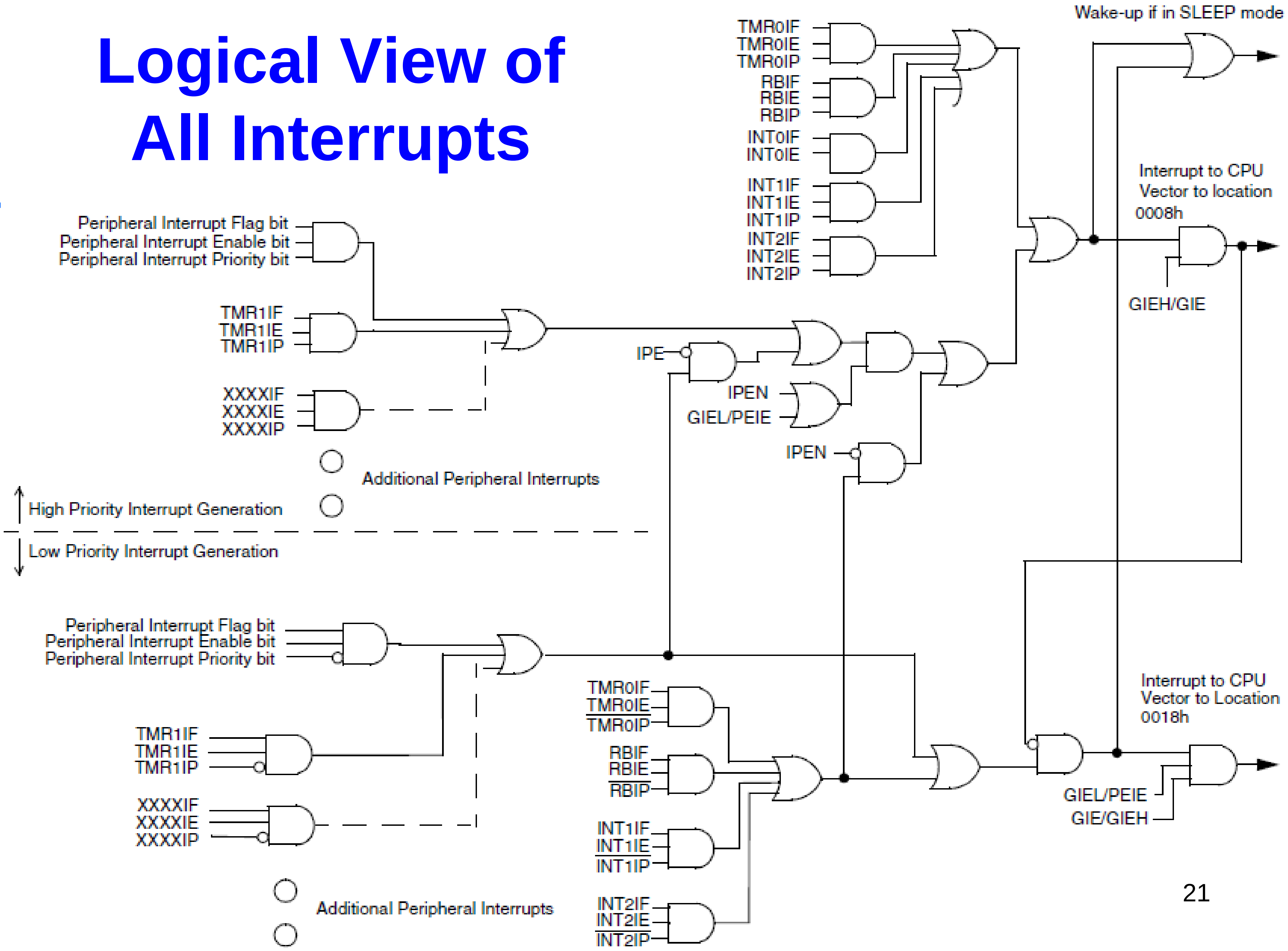
- Then we can assign low or high priority to interrupts by setting/clearing an interrupt priority bit in the IPRx SFRs
- This means there really are **three bits** controlling each interrupt
  - The INT0 (RB0) hardware interrupt can only be of high priority
- **Most importantly: When handling a low priority interrupt, high priority interrupts can steal the processor away**

CSE@UTA

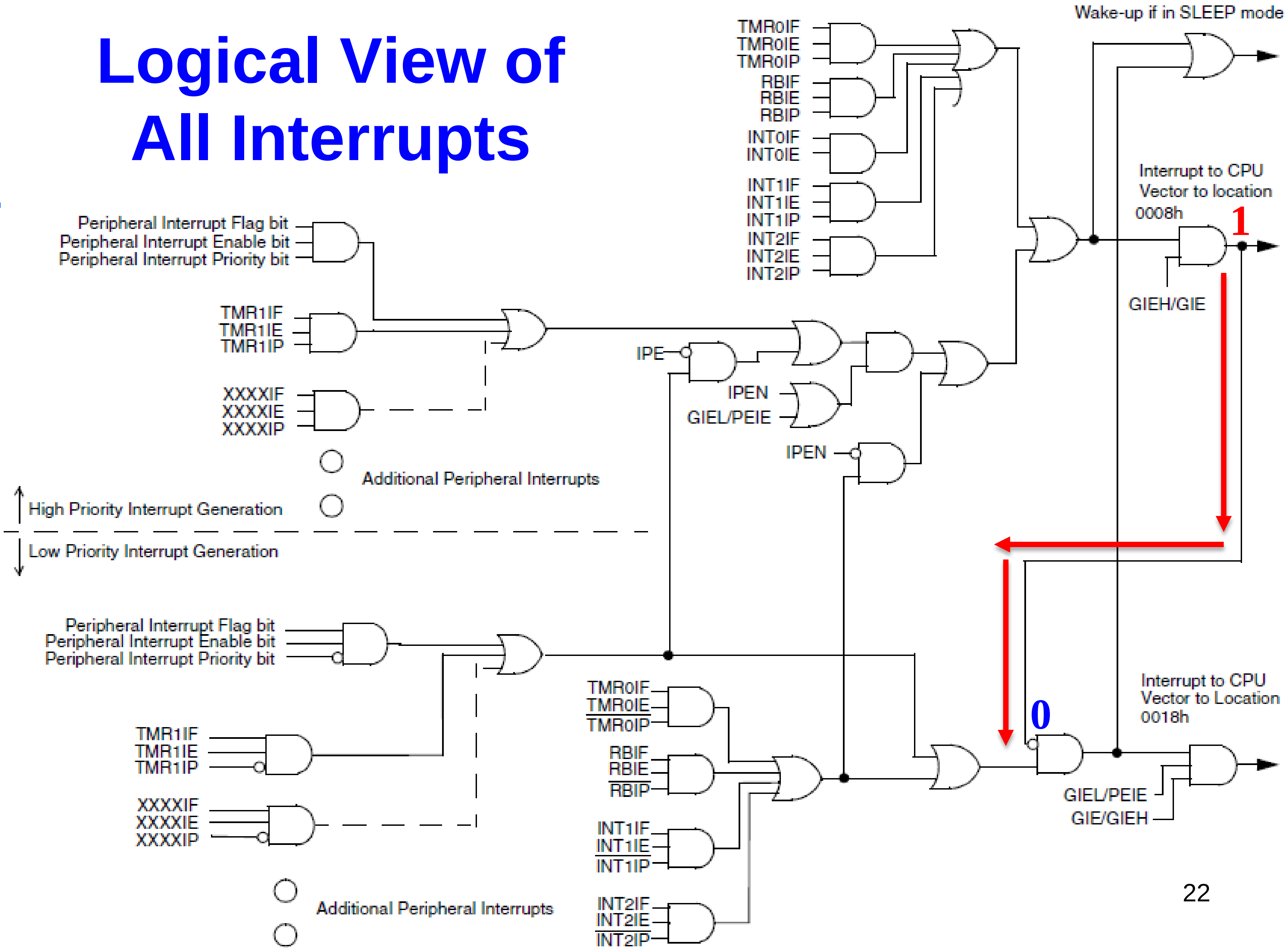
Logical View of High Priority Interrupts



Logical View of All Interrupts



Logical View of All Interrupts



CSE@UTA

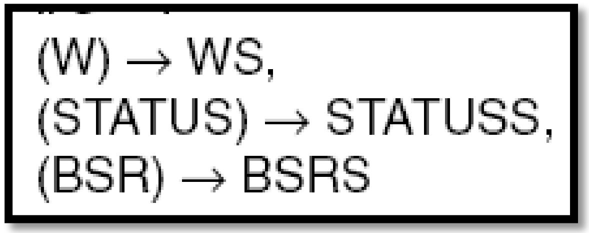
What Happens to Other Important Registers?

- What happens to other important registers (**WREG, Status, BSR**) that may be impacted by an interrupt
  - especially as they should be found the same way as they were left when returning
- The solution lies in the ISR having to save these registers at the beginning and restoring at the end
- **High-Priority**
  - **PIC18 automatically stores them in shadow registers**
  - **To restore registers use *RETFIE 1***
- **Low-Priority**
  - **Programmer must store them manually**

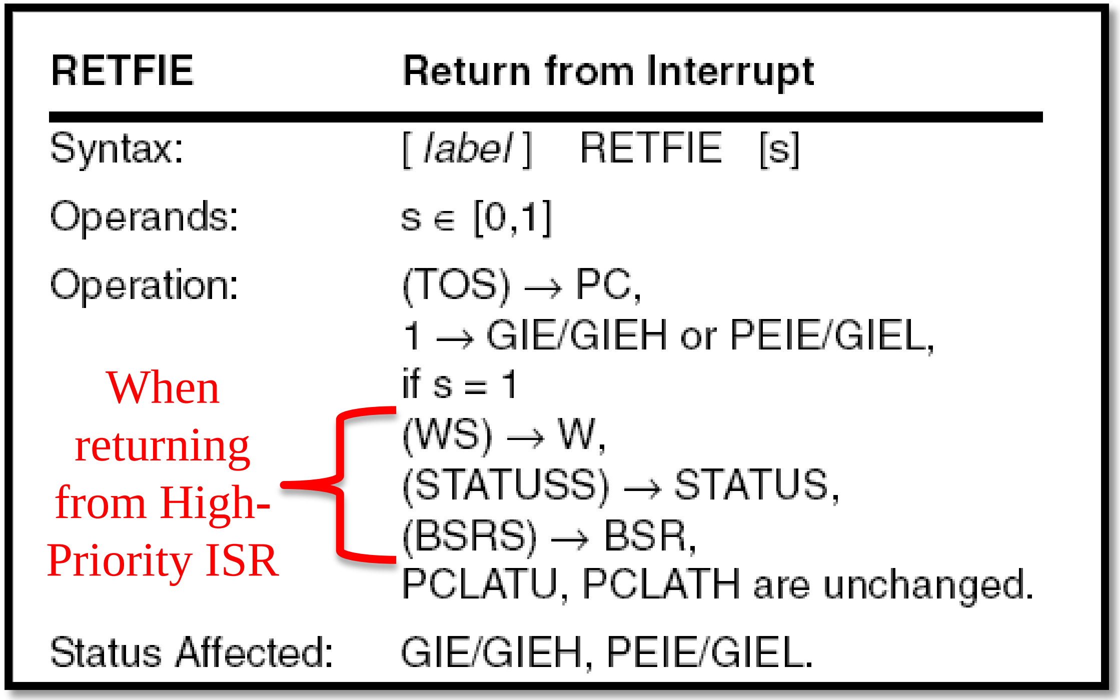
What Happens to Other Important Registers? – Fast Context Switching

- There is a **one-deep shadow register** set for WREG, Status, and BSR (similar to CALL and RETURN)

When jumping to High-Priority ISR



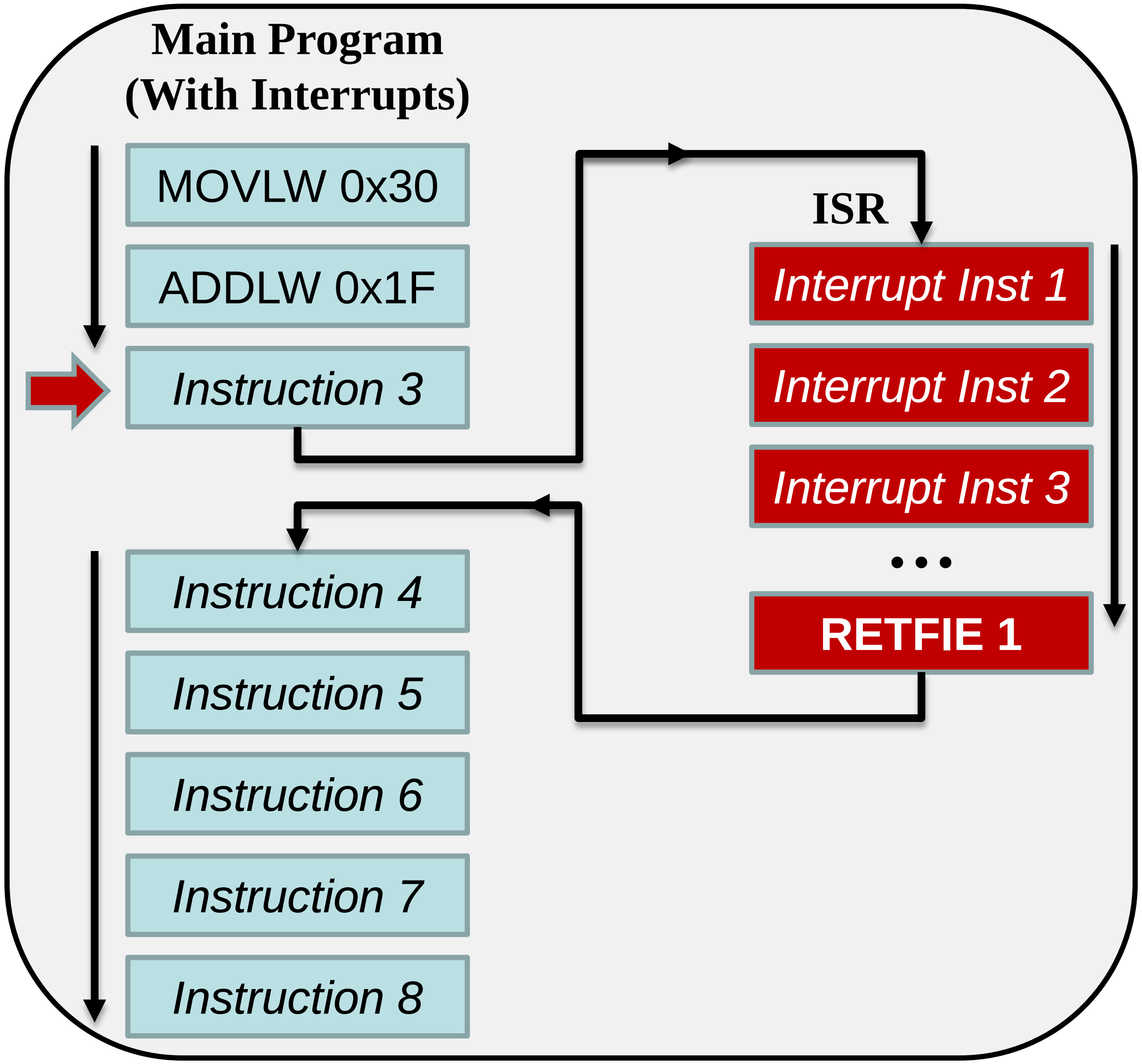
(W) → WS,
(STATUS) → STATUSS,
(BSR) → BSRS



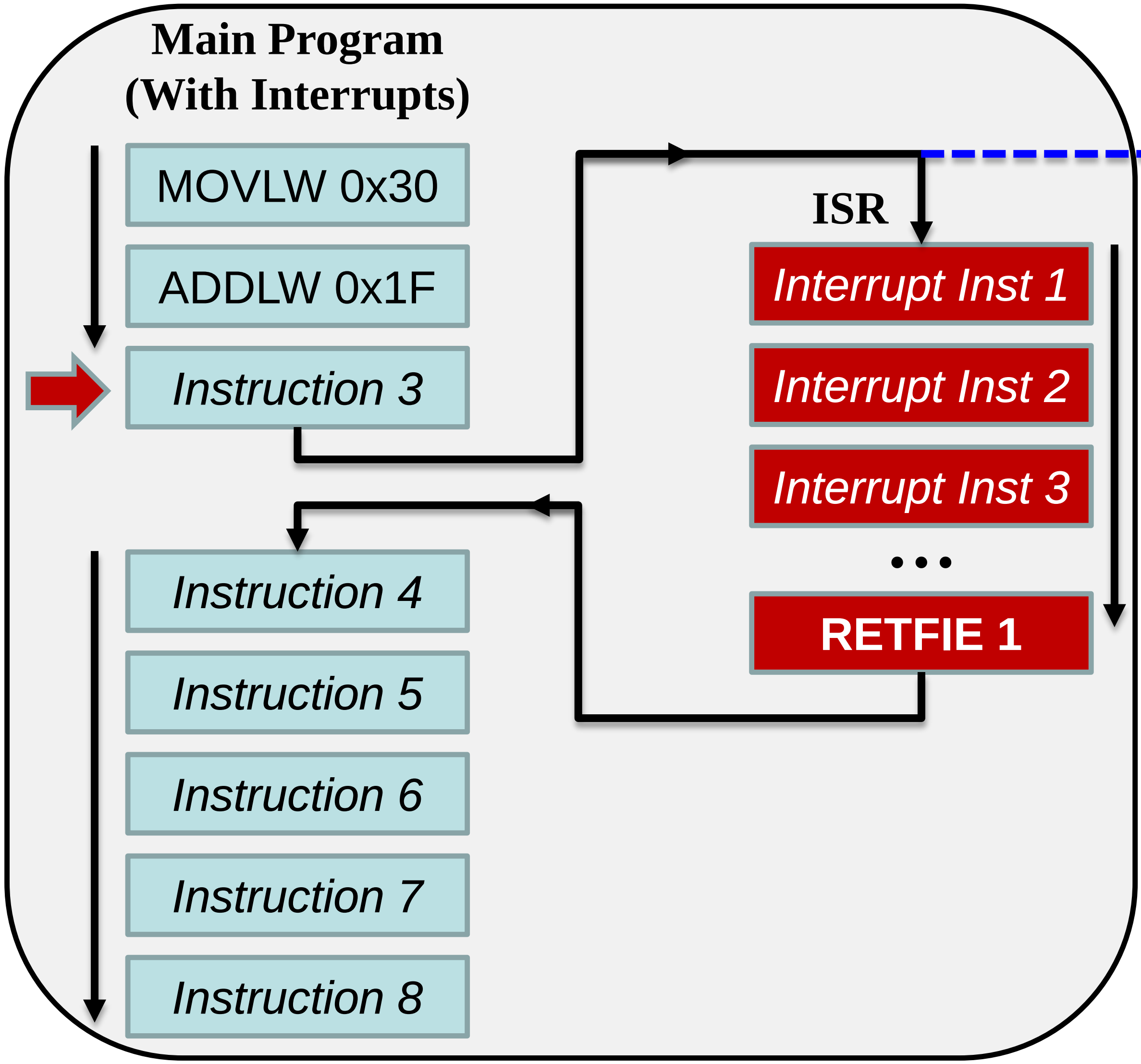
RETFIE	**Return from Interrupt**
Syntax:	[*label*] RETFIE [s]
Operands:	s ∈ [0,1]
Operation:	(TOS) → PC, 1 → GIE/GIEH or PEIE/GIEL, if s = 1 (WS) → W, (STATUSS) → STATUS, (BSRS) → BSR, PCLATU, PCLATH are unchanged.
Status Affected:	GIE/GIEH, PEIE/GIEL.

When returning from High-Priority ISR

Shadow Registers

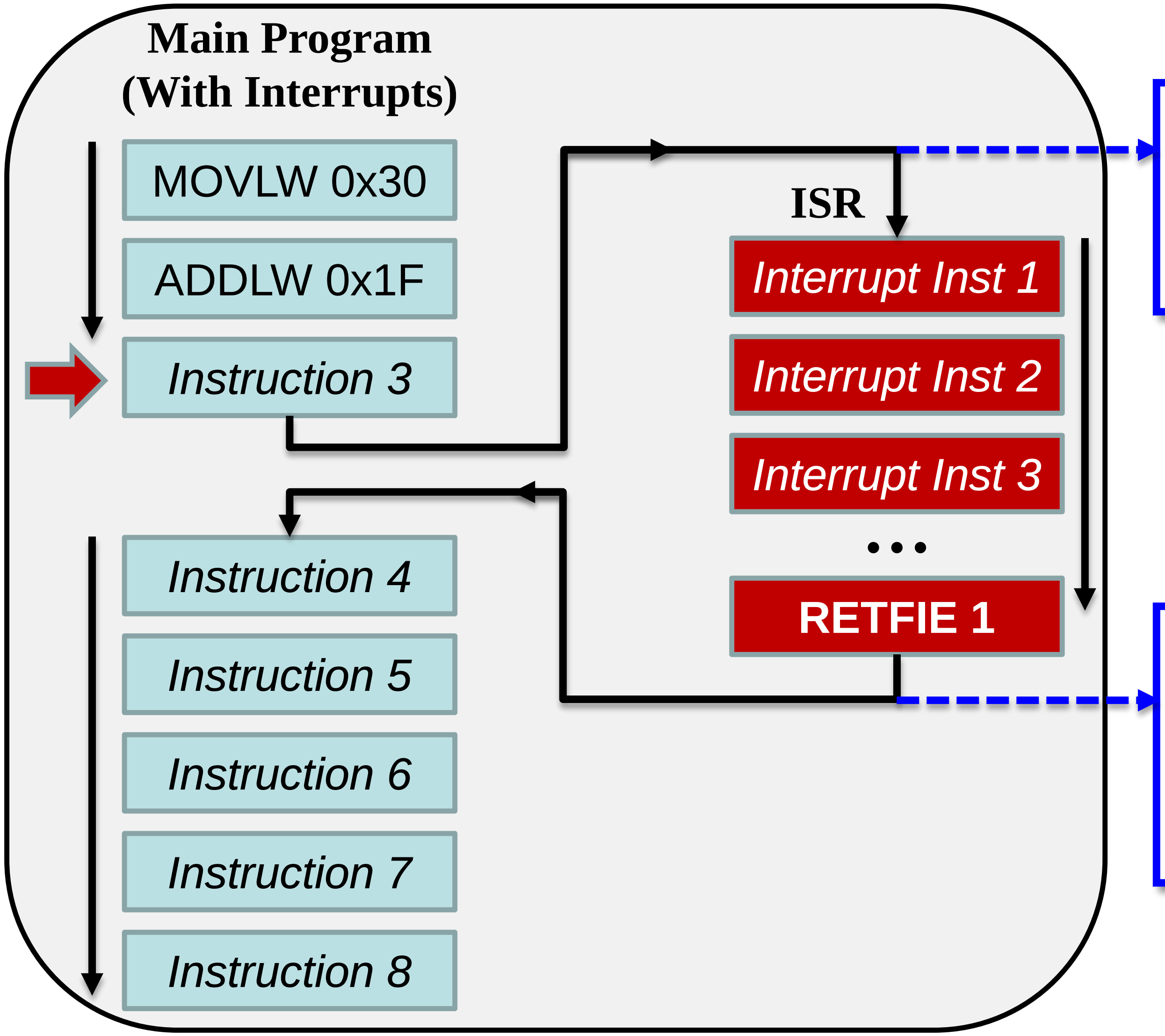


Shadow Registers



(W) → WS,
(STATUS) → STATUSS,
(BSR) → BSRS

Shadow Registers



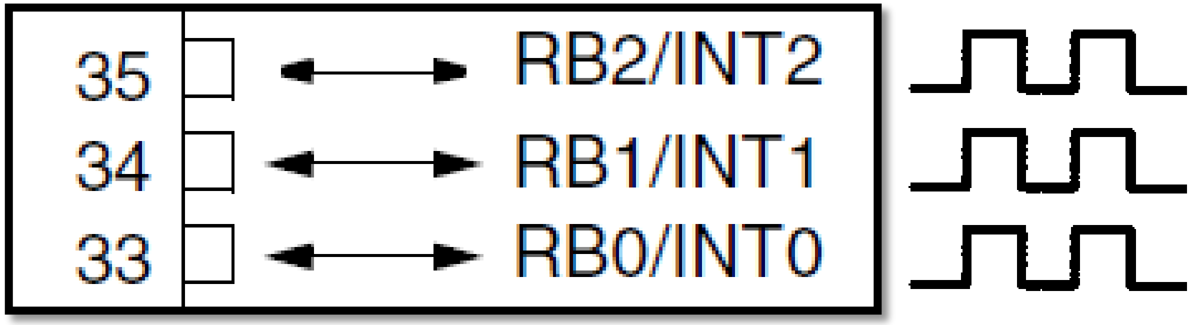
$(W) \rightarrow WS$,
$(STATUS) \rightarrow STATUSS$,
$(BSR) \rightarrow BSRS$

if $s = 1$
$(WS) \rightarrow W$,
$(STATUSS) \rightarrow STATUS$,
$(BSRS) \rightarrow BSR$,

CSE@UTA

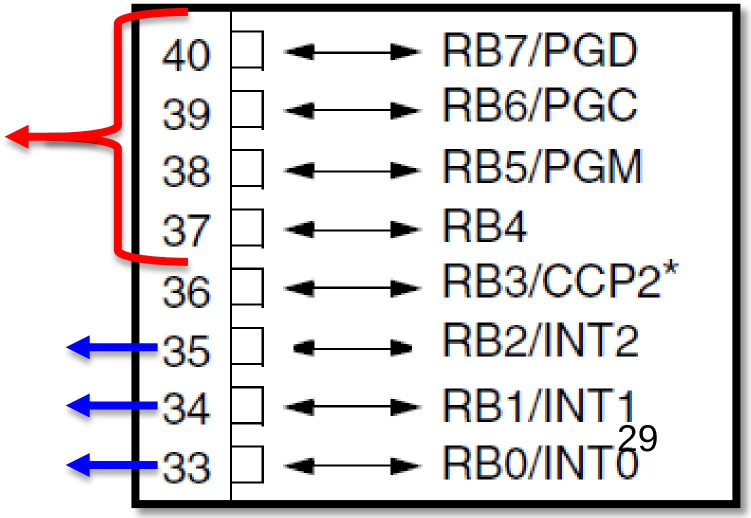
External INT Interrupts

- INT0, INT1, and INT2 are all interrupts assigned to digital I/O pins
  - To use them the corresponding TRISB bits have to be set
- INT interrupts are **edge triggered** (not level), thus a change must happen on the pins to trigger an interrupt
- Whether **rising** (default) or **falling edge** triggers the interrupt is software (INTCON2.INTEDGx bits) selectable
- When triggered (like many other flags) the ISR should explicitly clear the INTxIF flag
- INT0 is always of high priority, the other two can be set



External PORTB Interrupts

- Changes on RB4:RB7 can also cause interrupts but are on the <u>group of the bits</u> not individual
  - Leaves B3 as the only PORTB pin without interrupt capability
- Interrupt priority can be set HIGH or LOW
- When handling interrupt, PORTB should be read and INTCON.RBIF should be cleared
- Great for…
  - keyboard interfacing
  - grouped input
  - parallel input



INTCON

INTCON 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
bit 7							bit 0

bit 7 **GIE/GIEH:** Global Interrupt Enable bit

When IPEN = 0:

1 = Enables all unmasked interrupts

0 = Disables all interrupts

When IPEN = 1:

1 = Enables all high priority interrupts

0 = Disables all interrupts

bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit

When IPEN = 0:

1 = Enables all unmasked peripheral interrupts

0 = Disables all peripheral interrupts

When IPEN = 1:

1 = Enables all low priority peripheral interrupts

0 = Disables all low priority peripheral interrupts

bit 5 **TMR0IE:** TMR0 Overflow Interrupt Enable bit

1 = Enables the TMR0 overflow interrupt

0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE:** INT0 External Interrupt Enable bit

1 = Enables the INT0 external interrupt

0 = Disables the INT0 external interrupt

bit 3 **RBIE:** RB Port Change Interrupt Enable bit

1 = Enables the RB port change interrupt

0 = Disables the RB port change interrupt

bit 2 **TMR0IF:** TMR0 Overflow Interrupt Flag bit

1 = TMR0 register has overflowed (must be cleared in software)

0 = TMR0 register did not overflow

bit 1 **INT0IF:** INT0 External Interrupt Flag bit

1 = The INT0 external interrupt occurred (must be cleared in software)

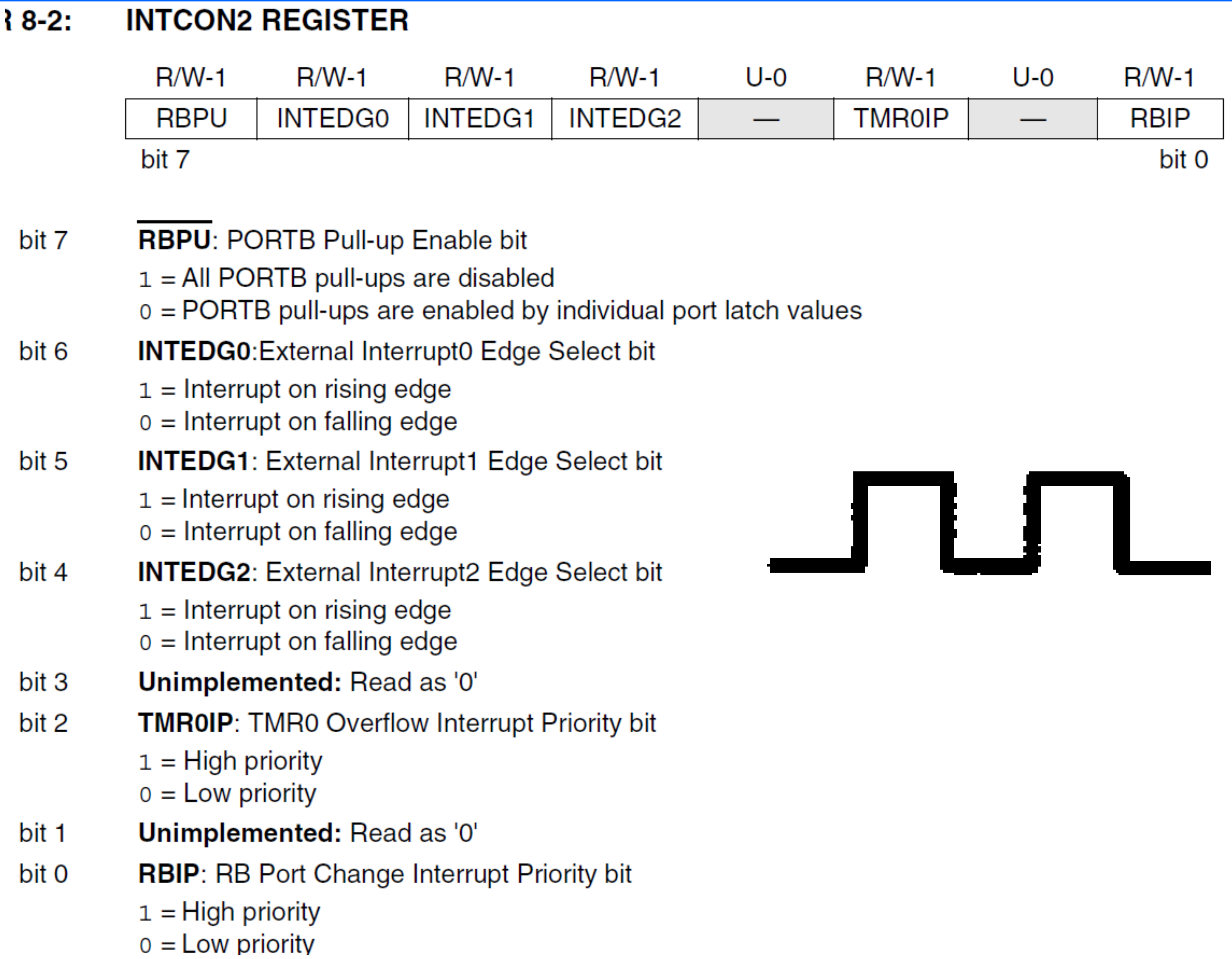
0 = The INT0 external interrupt did not occur

bit 0 **RBIF:** RB Port Change Interrupt Flag bit

1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)

0 = None of the RB7:RB4 pins have changed state

INTCON2



8-2: INTCON2 REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = All PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG0**:External Interrupt0 Edge Select bit
1 = Interrupt on rising edge
0 = Interrupt on falling edge

bit 5 **INTEDG1**: External Interrupt1 Edge Select bit
1 = Interrupt on rising edge
0 = Interrupt on falling edge

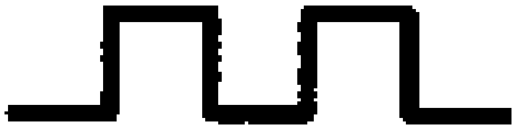
bit 4 **INTEDG2**: External Interrupt2 Edge Select bit
1 = Interrupt on rising edge
0 = Interrupt on falling edge

bit 3 **Unimplemented:** Read as '0'

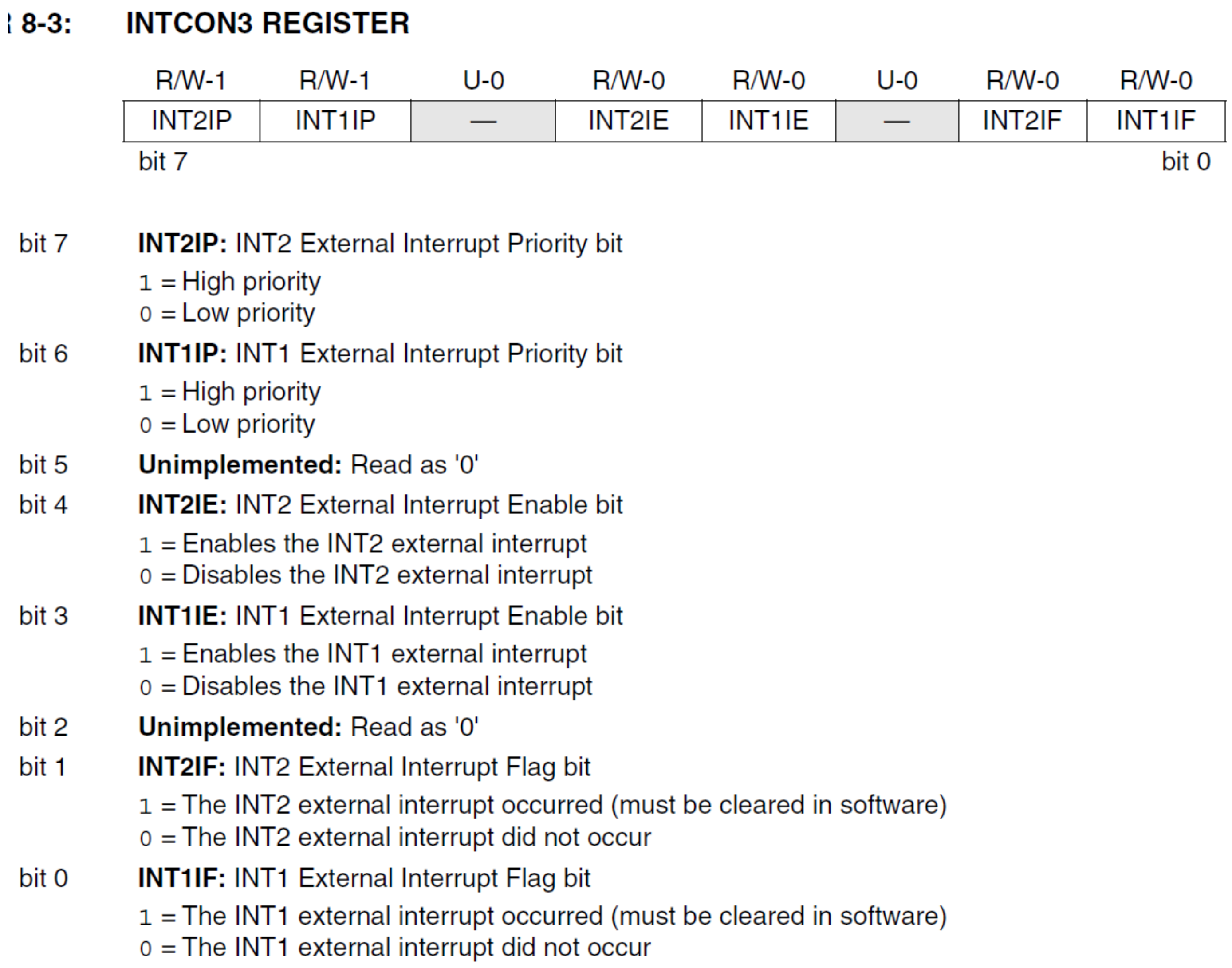
bit 2 **TMR0IP**: TMR0 Overflow Interrupt Priority bit
1 = High priority
0 = Low priority

bit 1 **Unimplemented:** Read as '0'

bit 0 **RBIP**: RB Port Change Interrupt Priority bit
1 = High priority
0 = Low priority



INTCON3



8-3: **INTCON3 REGISTER**

R/W-1	R/W-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF
bit 7							bit 0

bit 7 **INT2IP:** INT2 External Interrupt Priority bit
1 = High priority
0 = Low priority

bit 6 **INT1IP:** INT1 External Interrupt Priority bit
1 = High priority
0 = Low priority

bit 5 **Unimplemented:** Read as '0'

bit 4 **INT2IE:** INT2 External Interrupt Enable bit
1 = Enables the INT2 external interrupt
0 = Disables the INT2 external interrupt

bit 3 **INT1IE:** INT1 External Interrupt Enable bit
1 = Enables the INT1 external interrupt
0 = Disables the INT1 external interrupt

bit 2 **Unimplemented:** Read as '0'

bit 1 **INT2IF:** INT2 External Interrupt Flag bit
1 = The INT2 external interrupt occurred (must be cleared in software)
0 = The INT2 external interrupt did not occur

bit 0 **INT1IF:** INT1 External Interrupt Flag bit
1 = The INT1 external interrupt occurred (must be cleared in software)
0 = The INT1 external interrupt did not occur

PIR1

REGISTER 8-4: PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

bit 7 **PSPIF(1)**: Parallel Slave Port Read/Write Interrupt Flag bit
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred

bit 6 **ADIF**: A/D Converter Interrupt Flag bit
1 = An A/D conversion completed (must be cleared in software)
0 = The A/D conversion is not complete

bit 5 **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer, RCREG, is full (cleared when RCREG is read)
0 = The USART receive buffer is empty

bit 4 **TXIF**: USART Transmit Interrupt Flag bit (see Section 16.0 for details on TXIF functionality)
1 = The USART transmit buffer, TXREG, is empty (cleared when TXREG is written)
0 = The USART transmit buffer is full

bit 3 **SSPIF**: Master Synchronous Serial Port Interrupt Flag bit
1 = The transmission/reception is complete (must be cleared in software)
0 = Waiting to transmit/receive

bit 2 **CCP1IF**: CCP1 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode

bit 1 **TMR2IF:** TMR2 to PR2 Match Interrupt Flag bit
1 = TMR2 to PR2 match occurred (must be cleared in software)
0 = No TMR2 to PR2 match occurred

bit 0 **TMR1IF:** TMR1 Overflow Interrupt Flag bit
1 = TMR1 register overflowed (must be cleared in software)
0 = MR1 register did not overflow

PIR2

REGISTER 8-5: PIR2: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 2

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	EEIF	BCLIF	LVDIF	TMR3IF	CCP2IF
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **EEIF**: Data EEPROM/FLASH Write Operation Interrupt Flag bit
1 = The Write operation is complete (must be cleared in software)
0 = The Write operation is not complete, or has not been started

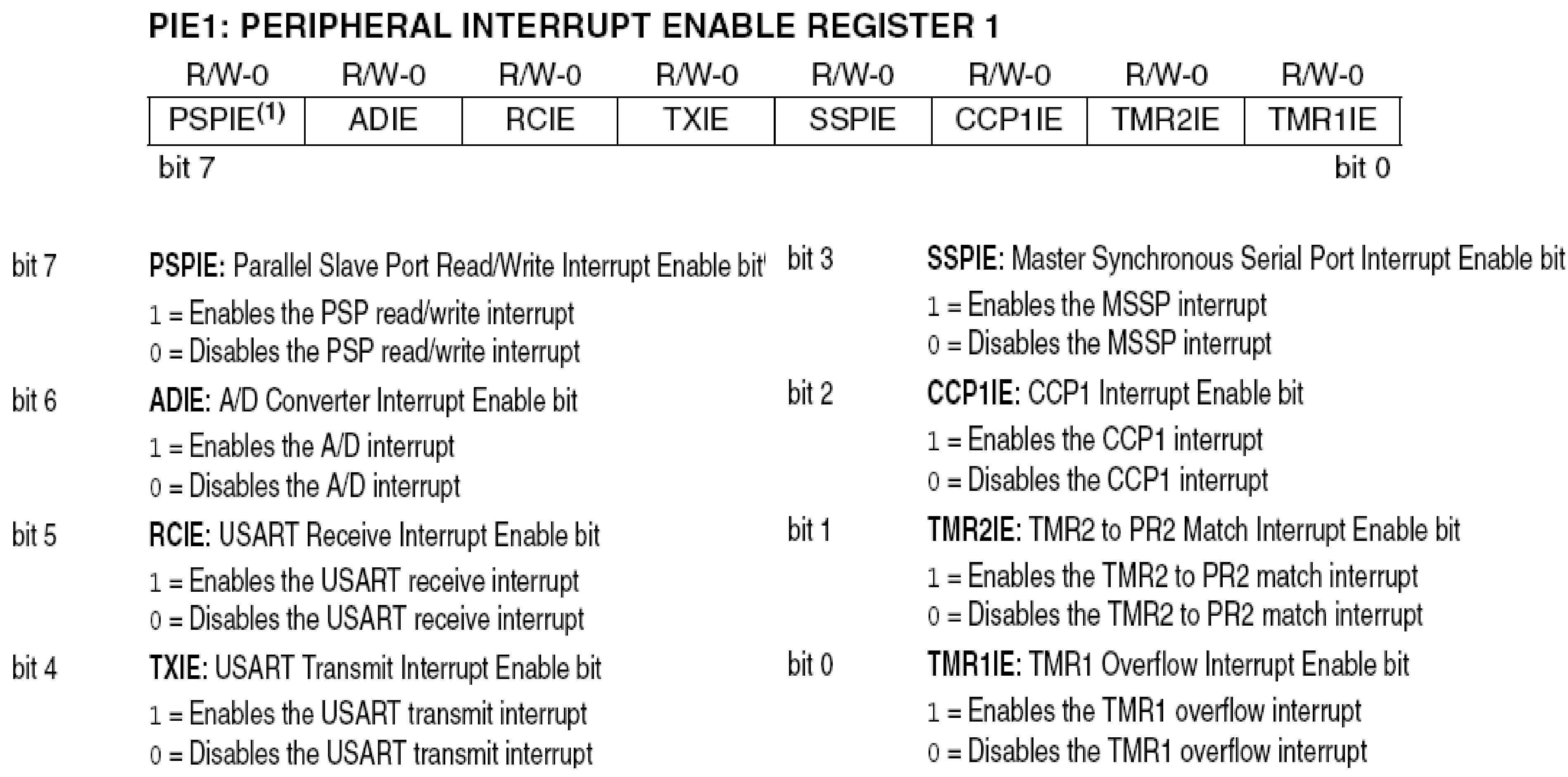
bit 3 **BCLIF**: Bus Collision Interrupt Flag bit
1 = A bus collision occurred (must be cleared in software)
0 = No bus collision occurred

bit 2 **LVDIF**: Low Voltage Detect Interrupt Flag bit
1 = A low voltage condition occurred (must be cleared in software)
0 = The device voltage is above the Low Voltage Detect trip point

bit 1 **TMR3IF**: TMR3 Overflow Interrupt Flag bit
1 = TMR3 register overflowed (must be cleared in software)
0 = TMR3 register did not overflow

bit 0 **CCP2IF**: CCPx Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode

PIE1



PIE1: PERIPHERAL INTERRUPT ENABLE REGISTER 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE[(1)]	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

bit 7 **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit[(1)]
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt

bit 6 **ADIE:** A/D Converter Interrupt Enable bit
1 = Enables the A/D interrupt
0 = Disables the A/D interrupt

bit 5 **RCIE:** USART Receive Interrupt Enable bit
1 = Enables the USART receive interrupt
0 = Disables the USART receive interrupt

bit 4 **TXIE:** USART Transmit Interrupt Enable bit
1 = Enables the USART transmit interrupt
0 = Disables the USART transmit interrupt

bit 3 **SSPIE:** Master Synchronous Serial Port Interrupt Enable bit
1 = Enables the MSSP interrupt
0 = Disables the MSSP interrupt

bit 2 **CCP1IE:** CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE:** TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

bit 0 **TMR1IE:** TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

CSE@UTA

PIE2

REGISTER 8-7: PIE2: PERIPHERAL INTERRUPT ENABLE REGISTER 2

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	EEIE	BCLIE	LVDIE	TMR3IE	CCP2IE
bit 7							bit 0

bit 7-5	**Unimplemented:** Read as '0'
bit 4	**EEIE**: Data EEPROM/FLASH Write Operation Interrupt Enable bit 1 = Enabled 0 = Disabled
bit 3	**BCLIE**: Bus Collision Interrupt Enable bit 1 = Enabled 0 = Disabled
bit 2	**LVDIE**: Low Voltage Detect Interrupt Enable bit 1 = Enabled 0 = Disabled
bit 1	**TMR3IE**: TMR3 Overflow Interrupt Enable bit 1 = Enables the TMR3 overflow interrupt 0 = Disables the TMR3 overflow interrupt
bit 0	**CCP2IE**: CCP2 Interrupt Enable bit 1 = Enables the CCP2 interrupt 0 = Disables the CCP2 interrupt

CSE@UTA

IPR1

REGISTER 8-8: IPR1: PERIPHERAL INTERRUPT PRIORITY REGISTER 1

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
PSPIP(1)	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP
bit 7							bit 0

bit 7 **PSPIP(1)**: Parallel Slave Port Read/Write Interrupt Priority bit
1 = High priority
0 = Low priority

bit 6 **ADIP**: A/D Converter Interrupt Priority bit
1 = High priority
0 = Low priority

bit 5 **RCIP**: USART Receive Interrupt Priority bit
1 = High priority
0 = Low priority

bit 4 **TXIP**: USART Transmit Interrupt Priority bit
1 = High priority
0 = Low priority

bit 3 **SSPIP**: Master Synchronous Serial Port Interrupt Priority bit
1 = High priority
0 = Low priority

bit 2 **CCP1IP**: CCP1 Interrupt Priority bit
1 = High priority
0 = Low priority

bit 1 **TMR2IP**: TMR2 to PR2 Match Interrupt Priority bit
1 = High priority
0 = Low priority

bit 0 **TMR1IP**: TMR1 Overflow Interrupt Priority bit
1 = High priority
0 = Low priority

CSE@UTA

IPR2

REGISTER 8-9: IPR2: PERIPHERAL INTERRUPT PRIORITY REGISTER 2

U-0	U-0	U-0	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
—	—	—	EEIP	BCLIP	LVDIP	TMR3IP	CCP2IP
bit 7							bit 0

bit 7-5	**Unimplemented:** Read as '0'
bit 4	**EEIP**: Data EEPROM/FLASH Write Operation Interrupt Priority bit 1 = High priority 0 = Low priority
bit 3	**BCLIP**: Bus Collision Interrupt Priority bit 1 = High priority 0 = Low priority
bit 2	**LVDIP**: Low Voltage Detect Interrupt Priority bit 1 = High priority 0 = Low priority
bit 1	**TMR3IP**: TMR3 Overflow Interrupt Priority bit 1 = High priority 0 = Low priority
bit 0	**CCP2IP**: CCP2 Interrupt Priority bit 1 = High priority 0 = Low priority

RCON

REGISTER 8-10: RCON REGISTER

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

bit 7 **IPEN:** Interrupt Priority Enable bit
`1` = Enable priority levels on interrupts
`0` = Disable priority levels on interrupts (16CXXX Compatibility mode)

bit 6-5 **Unimplemented:** Read as '0'

bit 4 **$\overline{RI}$:** `RESET` Instruction Flag bit
For details of bit operation, see Register 4-3

bit 3 **$\overline{TO}$:** Watchdog Time-out Flag bit
For details of bit operation, see Register 4-3

bit 2 **$\overline{PD}$:** Power-down Detection Flag bit
For details of bit operation, see Register 4-3

bit 1 **$\overline{POR}$:** Power-on Reset Status bit
For details of bit operation, see Register 4-3

bit 0 **$\overline{BOR}$:** Brown-out Reset Status bit
For details of bit operation, see Register 4-3

Interrupt Programming from Assembly

- Really just a quasi-tedious job of setting the right bits in the right registers and “org”-ing the code at the right place

```
        ORG 0000H
        GOTO MAIN

        ORG 0008H
        GOTO HP_ISR

        ORG 00018H
        GOTO LP_ISR
…
…
BSF    INTCON, INT0IE
BCF    INTCON2, INTEDG0
BSF    INTCON, GIE
```

```
                ORG     200H
HP_ISR          BTFSS   INTCON, INT0IF
                RETFIE  1
                BTG     PORTB, 7
                BCF     INTCON, INT0IF
                RETFIE  1
```

Interrupt Programming from C

- We need to define functions that are for **high priority** and **low priority** ISRs
- We need to make sure that our ISRs are in the right place
- We **do not** need to worry about **context-switching**, the C compiler is going to make sure our registers are properly handled and variables that need saving are saved
- Interrupt handlers should start off with an “if” or a “switch-case” complex to **identify the source of the interrupt**

CSE@UTA

Defining ISRs in C18

- Defining functions that are for high priority and low priority ISRs:
  - At the beginning of the program, have a prototype of all functions (including ISRs)
  - Use ***#pragma interrupt function_name*** and ***#pragma interruptlow function_name*** to tell C18 compiler that a function is an interrupt function (so it can use proper RETFIE returns and fast context switching)

CSE@UTA

Placing ISRs in C18

- Make sure that our ISRs are in the right place
- At the beginning of the code insert goto instructions to the interrupt vectors
- Use ASM to limit size and ensure it fits in ROM

```
#pragma code My_Hi_Priority_Int = 0x0008
void My_Hi_Priority_Int(void)
{
    _asm
          GOTO chk_isr
    _endasm
}
```

Interrupt Handling in C18

```
#include <P18F452.h>

void My_ISR_High(void);
void My_ISR_Low(void);

#pragma code My_Hi_Priority_Int = 0x0008
void My_Hi_Priority_Int(void)
{
      _asm
          GOTO My_ISR_High
      _endasm
}

#pragma code My_Lo_Priority_Int = 0x00018
void My_Lo_Priority_Int(void)
{
      _asm
          GOTO My_ISR_Low
      _endasm
}
```

```
void main()
{
      //main control code
      //and Interrupt Settings
      ...
}

//other functions
...


#pragma interrupt My_ISR_High
void My_ISR_High(void)
{
      //interrupt handling for HIGH
      ...
}

#pragma interruptlow My_ISR_Low
void My_ISR_Low(void)
{
      //interrupt handling for LOW
      ...
}
```

CSE@UTA

Placing ISRs in XC8

- Much simpler in XC8 compiler
- Only need to know two keywords
  - "interrupt" and "low_priority"

- **High-Priority**

```
void interrupt My_ISR_High(void)
{
        //interrupt handling for HP
}
```

- **Low-Priority**

```
void interrupt low_priority My_ISR_Low(void)
{
        //interrupt handling for LP
}
```

Interrupt Handling in XC8

```
#include <P18F452.h>

void My_ISR_High(void);
void My_ISR_Low(void);

void main()
{
      //main control code
      //and Interrupt Settings
      ...
}

//other functions
...
```

//placing of interrupt code at the correct locations is automatically handled by the XC8 compiler

```
void interrupt My_ISR_High(void)
{
      //interrupt handling for HIGH
      if(INT0IF == 1 && INT0IE == 1)
               //INT0 interrupt tripped
      if(TMR0IF == 1 && TMR0IE == 1)
               //Timer 0 interrupt tripped
      ...
}


void interrupt low_priority My_ISR_Low(void)
{
      //interrupt handling for LOW
      if(ADIF == 1 && ADIE == 1)
               //ADC conversion done
      if(INT1IF == 1 && INT1IE == 1)
               //INT1 interrupt tripped
      if(RCIF == 1 && RCIE == 1)
               //Serial reception occurred
      ...
}
```

ADC Interrupt Example

```
#include <P18F452.h>
void My_ISR_High(void);

void main()
{
      ADCON1 = 0b11001110; //ADC settings
      ADCON0 = 0b10000001;
      PIR1bits.ADIF = 0; //Clear ADIF flag bit
      IPR1bits.ADIP = 1; //ADC is HIGH Priority
      PIE1bits.ADIE = 1; //Set ADIE enable bit
      INTCONbits.PEIE = 1; //Set PEIE enable bit
      INTCONbits.GIE = 1; //Set GIE enable bit

      while(1)
      {
          ADCON0bits.GO = 1; //Start ADC
          ...// go on with other code
          ...
      }
}
```

```
void interrupt My_ISR_High(void)
{
      //interrupt handling for HIGH
      if(INT0IF == 1 && INT0IE == 1)
               //INT0 interrupt tripped
      if(ADIF == 1 && ADIE == 1)
      {
        //ADC conversion done
        //Get result from ADRESH/L
        PIR1bits.ADIF = 0; //clear flag
      }
}
```

Summary

- Interrupts are a great way to handle peripheral attention or external happenings
- Some of the most used interrupts are timers (later), external hardware, serial communications, and ADC ready
- All interrupts in the PIC18 can be masked in a group or individually
- We can have two levels of priorities, with an almost fully configurable what interrupt belong to what level relationship
- Programming ISRs from C requires knowledge of how the compiler is told about ISRs
  - Consult the compiler's user guide for specifics