SNS COLLEGE OF TECHNOLOGY

(An Autonomous Institution)

COIMBATORE-35

Accredited by NBA-AICTE and Accredited by NAAC – UGC with A++ Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

COURSE NAME: 19EEB303 / Microcontroller and its Applications

III YEAR / VI SEMESTER

Unit II – **PIC MICROCONTROLLER**

Topic : Interrupts

INTERRUPTS

- Interrupts are basically internal/external signals that suspend the main routine being done/executed. While reading this article, your main routine is “Reading The Tutorial”. This main routine could be interrupted by many sudden events. If your phone suddenly started ringing during the “Reading” process, The main routine “Reading” will be suspended “Interrupted”.

INTERRUPTS

Interrupts could be classified based on the source of the interrupt signal, and also based on the way it's implemented in memory. Interrupt signals could be generated by hardware or software. Interrupts could be implemented in memory as vectored or non-vectored interrupts. So, let's classify interrupts in more detail.

Software Interrupts

- The programmer (you and me) can purposely fire an interrupt signal whenever he wants using specific instructions (e.g. **SWI**). This type of interrupt signals is said to be software interrupts. Because the signal's source is a software instruction. We may not use such a thing during this series of tutorials. However, you should know that many CPUs has specific instructions that generate a software interrupt signal.

INTERRUPTS

Hardware Interrupts

Almost all the peripherals/modules within a microcontroller generate interrupt signals to indicate various events. That's why this tutorial precedes most of the upcoming modules. Such as Timers, CCP, SPI, UART, I2C, ADC, EEPROM, etc.. All of these modules generate interrupt signals to indicate starting, termination or failure of their current operation.

- You should also note that hardware interrupts could be internally or externally triggered. That's why many authors in our literature call internal modules interrupts "**Internal**", and the externally triggered interrupts "**External**" interrupts. Anyway, hardware interrupts are the most common ones, and it's our job to efficiently handle them!

INTERRUPTS

Vectored Interrupts

For vectored interrupts, the CPU knows exactly the address of the ISR handler. The CPU has these addresses pre-defined in memory (interrupt table) in advance. When a vectored interrupt is fired, the interrupting module/device sends its specific vector to the CPU via the data bus. Then, the CPU will perform a look-up in the interrupt table in memory. And then it branches to the ISR handler code associated to the interrupting device and executes it.

Non-Vectored Interrupts

- For non-vectored interrupts, the CPU has a hardware fixed address called the **interrupt vector**. When an interrupt is fired, the CPU will push the PC to the stack. Then it'll jump to the interrupt vector address and then branches to the ISR handler code. Which is a hard-coded ISR in a specific portion of the memory.

INTERRUPTS

In the microcontroller we're using (PIC16F877A), the interrupts are non-vectored in memory. So there is a common interrupt vector @ the address 0004h, which is always **skipped over** while executing the firmware in the program memory. This process is indicated in the figure below

INTERRUPTS

As we previously mentioned, there is no marker to help the CPU keep track of the last instruction was being executed when the interruption occurred. That's why there is a hardware implemented program stack.

The program stack is basically an 8-levels (8-Registers) structure that holds the addresses of program instructions to be executed. It's a LIFO structure which means Last-In-First-Out, The last **PUSH**ed address is the first **POP**ed one. Before executing any instruction, its address is **PUSH**ed to the stack.

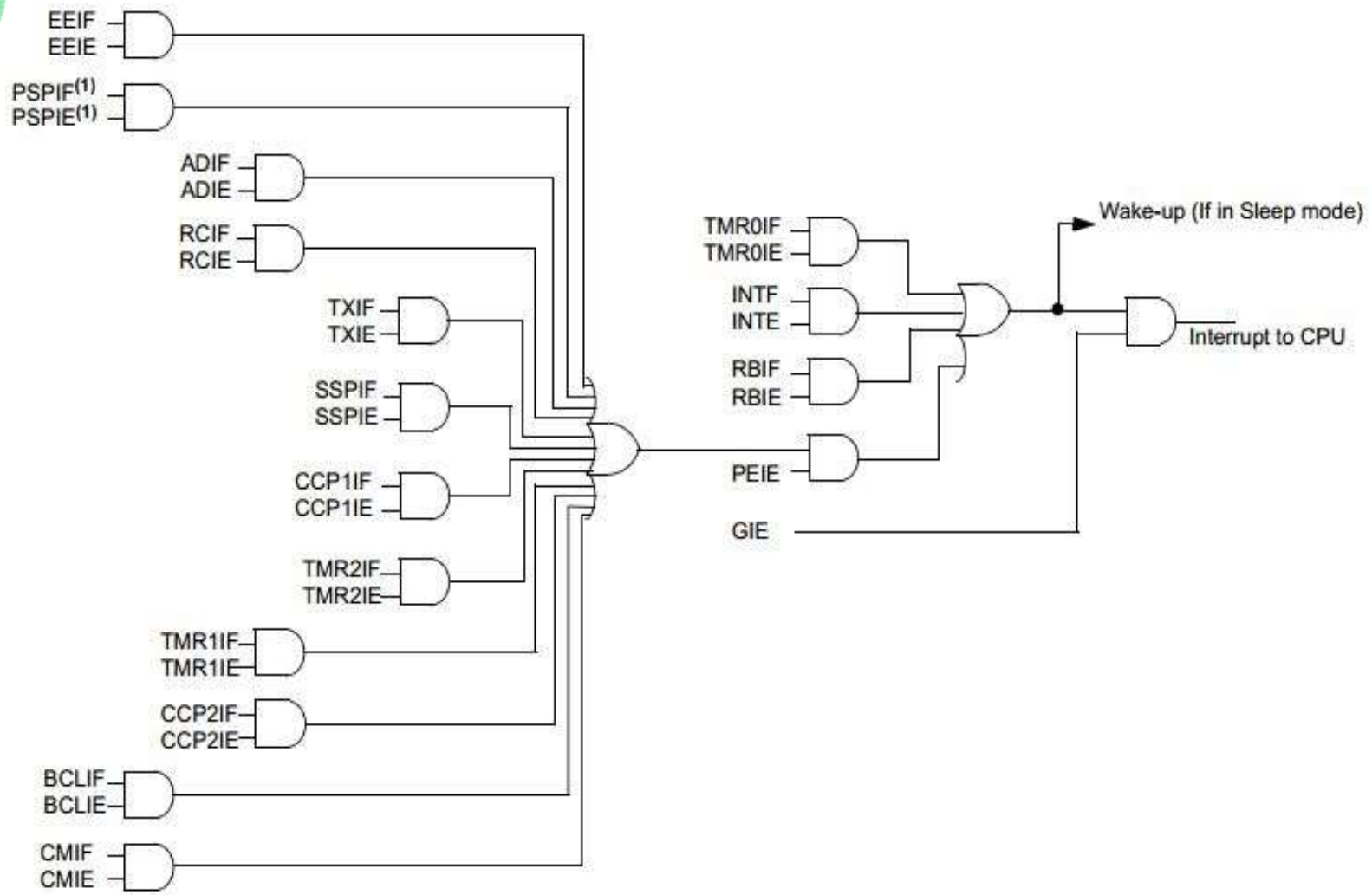
Whenever an interrupt occurs, the CPU will **PUSH** the next instruction's address in the stack. Then the address of the interrupt vector is automatically pushed to the PC so that the CPU branches directly to the ISR handler code to execute it. And when it's done with the ISR, the CPU will automatically perform the **RETFIE** instruction to return from the interrupt service routine to the main routine (program). Which obviously **POP**s the last instruction's address saved in the stack. Which was the next instruction to be executed before the interruption!

INTERRUPTS

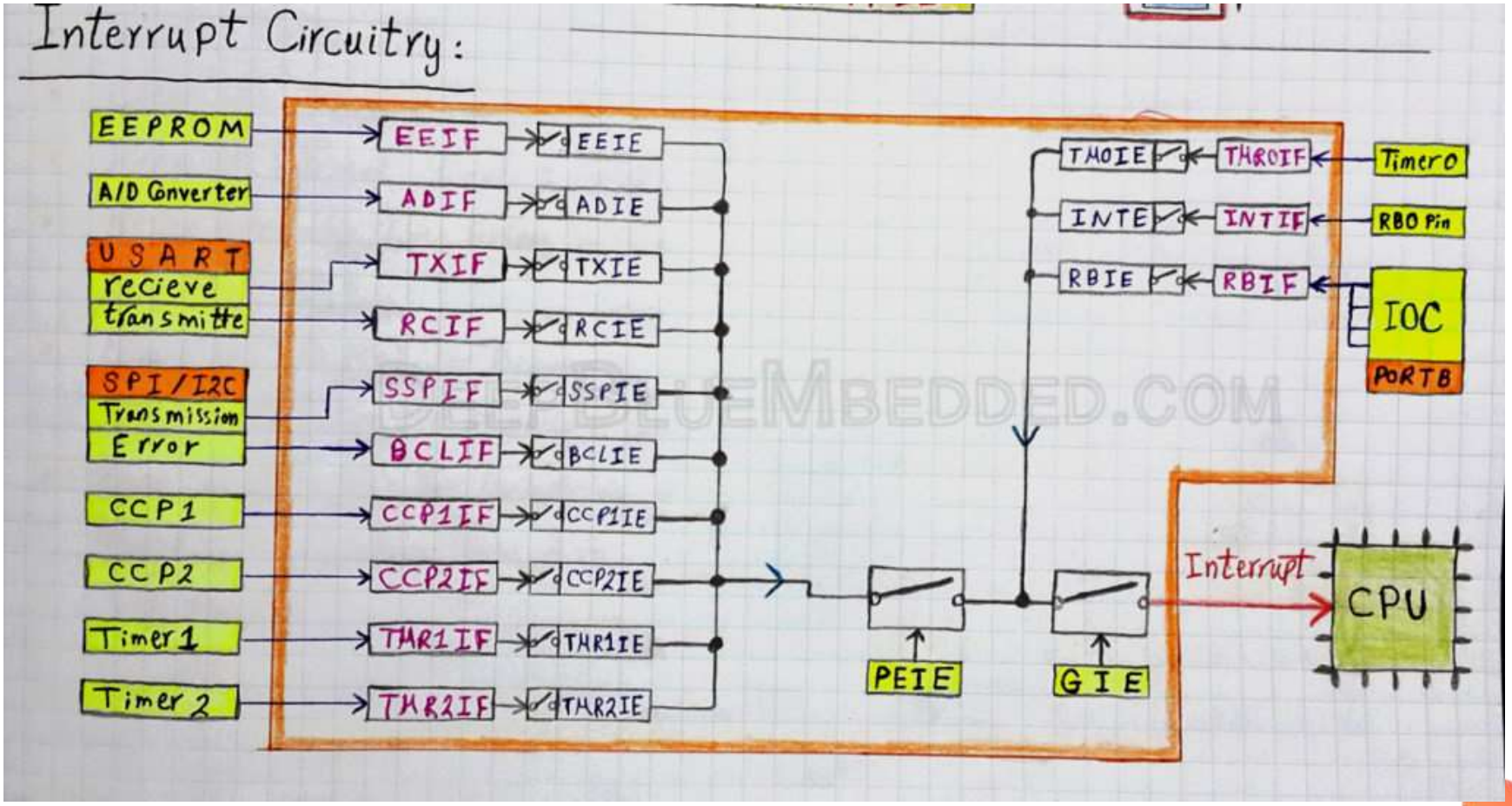
Interrupt Circuitry

- The interrupt circuitry is the digital logic circuit that drives the interruption systems within the microcontroller. We use this circuit for both configuring & handling interrupts. The diagram could be easily found in the datasheet

INTERRUPTS


EEIF
EEIE
PSPIF(1)
PSPIE(1)
ADIF
ADIE
RCIF
RCIE
TXIF
TXIE
SSPIF
SSPIE
CCP1IF
CCP1IE
TMR2IF
TMR2IE
TMR1IF
TMR1IE
CCP2IF
CCP2IE
BCLIF
BCLIE
CMIF
CMIE
TMR0IF
TMR0IE
INTF
INTE
RBIF
RBIE
PEIE
GIE
Wake-up (If in Sleep mode)
Interrupt to CPU

INTERRUPTS

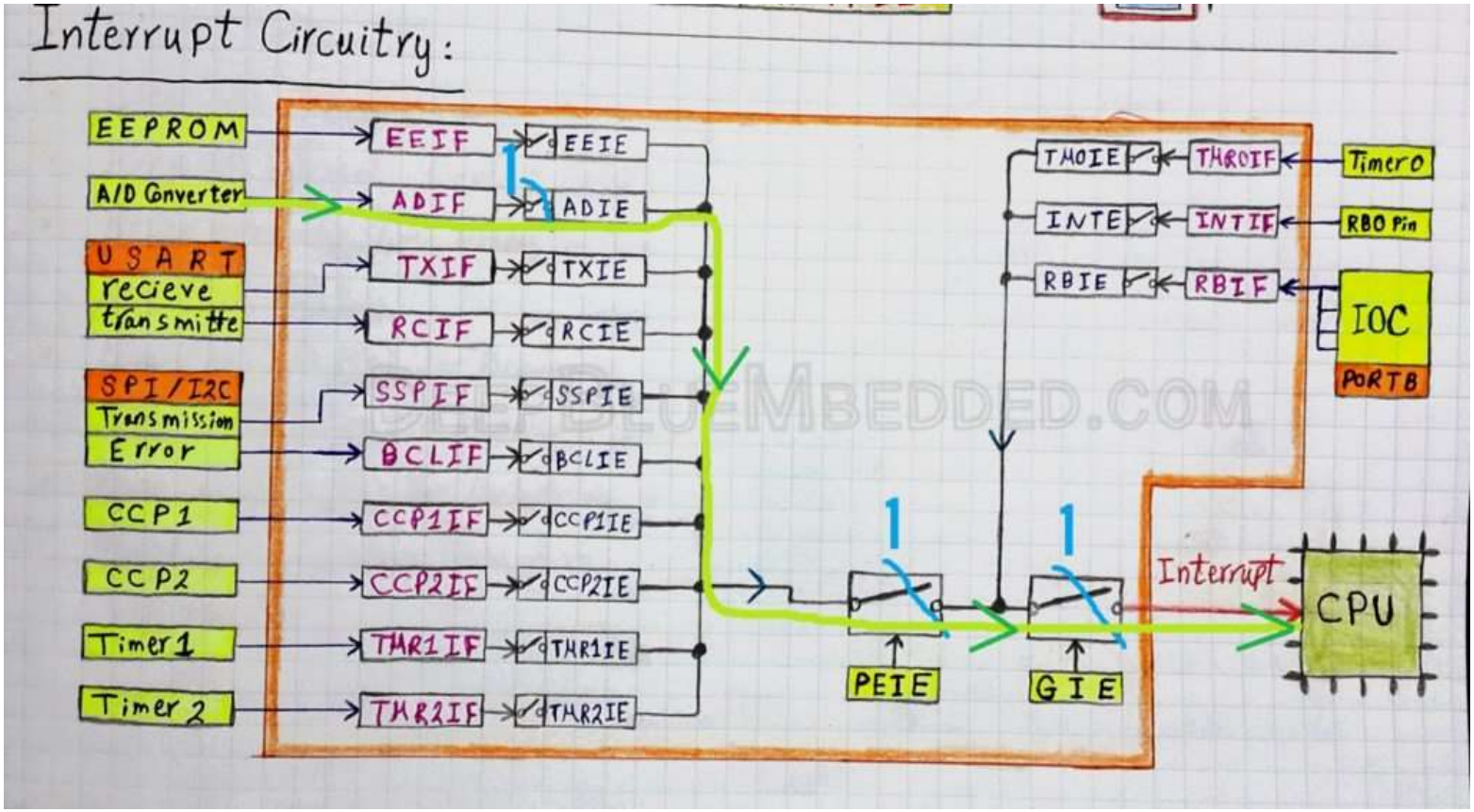

Interrupt Circuitry:
EEPROM
A/D Converter
USART
recieve
transmitte
SPI/I2C
Transmission
Error
CCP1
CCP2
Timer1
Timer2
EEIF
ADIF
TXIF
RCIF
SSPIF
BCLIF
CCP1IF
CCP2IF
THR1IF
THR2IF
EEIE
ADIE
TXIE
RCIE
SSPIE
BCLIE
CCP1IE
CCP2IE
THR1IE
THR2IE
PEIE
GIE
TMOIE
INTE
RBIE
T0IF
INTIF
RBIF
Timer0
RB0 Pin
IOC
PORTB
Interrupt
CPU

INTERRUPTS

There is only one wire coming out of the interrupt circuitry and heading to the CPU. This is because our interrupts are non-vectored so they do share one common interruption signal to the CPU which has no idea about the device that fired the interrupt signal.

- Configuring an interrupt source is the first step in working with interrupts. Enabling the ADC interrupt will result in firing an interrupt signal upon each successful ADC conversion process to notify the CPU about it. To enable the ADC interrupt signal you should obviously set all the bits in the way between that module & the CPU

INTERRUPTS


Interrupt Circuitry:
EEPROM
A/D Converter
USART
recieve
transmitte
SPI/I2C
Transmission
Error
CCP1
CCP2
Timer1
Timer2
EEIF
ADIF
TXIF
RCIF
SSPIF
BCLIF
CCP1IF
CCP2IF
THR1IF
THR2IF
EEIE
ADIE
TXIE
RCIE
SSPIE
BCLIE
CCP1IE
CCP2IE
THR1IE
THR2IE
PEIE
GIE
1
1
T0IE
INTE
RBIE
T0IF
INTIF
RBIF
Timer0
RB0 Pin
IOC
PORTB
Interrupt
CPU

INTERRUPTS

Set the **ADCIE**, **PEIE**, and **GIE** bits. Now, when an ADC conversion is complete. The ADCIF will be set, and the 1(High) signal will reach the CPU to notify it that an interrupt has occurred. But, the CPU now has no idea about which module has generated this signal.

There are interrupt flag bits, these bits are set upon devices' interruption respectively. If an ADC conversion is complete, the **ADCIF** flag bit is set. If Timer1 has reached overflow state, the **TMR1IF** flag bit is set. And so on.

- Upon receiving interrupt signal, the CPU PUSHes the machine state in the stack and branches to the ISR handler. In which we must first poll (check) the **Flag Bits**, in order to determine the interruption source to service it respectively.

INTERRUPTS

- The registers which control the interrupt circuitry are the following 5-Registers

INTCON	PIE1	PIE2	PIR1	PIR2

- These registers contain both the interrupt enable bits and the interrupt flag bits. For each interruption source (up to 15). In MPLAB XC8, we've bit-fields with the same names found in the datasheet for these bits.

INTERRUPTS

Registers Used to Process Interrupts

Interrupt Control Register

INTCON register

GIE - Global Interrupt Enable
PEIE - Peripheral Interrupt Enable
TMR0IE - Timer0 Interrupt Enable
INTE - External Interrupt Enable
IOCIE - Interrupt on Change Enable
TMR0IF - Timer0 Interrupt flag
INTF - External Interrupt flag
IOCIF - Interrupt on Change flag

INTCON contains global and peripheral interrupt enable flags as well as the individual interrupt request flags and interrupt enable flags for three of the PIC16F1xxxx interrupts.

INTERRUPTS

Interrupt Enable Registers

PIE1 register

MR1GIE - Timer1 Gate

Interrupt Enable

ADIE - Analog-to-Digital Converter (ADC) Interrupt Enable

RCIE - Universal Synchronous Asynchronous Receiver Transmitter (USART) Receive Interrupt Enable

TXIE - USART Transmit Interrupt Enable

SSPIE - Synchronous Serial Port (MSSP) Interrupt Enable

CCP1IE - CCP1 Interrupt Enable

TMR2IE - Timer2 Interrupt Enable

TMR1IE - Timer1 Interrupt Enable

INTERRUPTS

PIE2 register

OSFIE	C2IE	C1IE	EEIE	BCLIE	LCDIE	----	CCP2IE
bit 7							bit 0

OSFIE - Oscillator Fail Interrupt Enable
C2IE - Comparator C2 Interrupt Enable
C1IE - Comparator C1 Interrupt Enable
EEIE - EEPROM Write Completion Interrupt Enable
BCLIE - MSSP Bus Collision Interrupt Enable
LCDIE - LCD Module Interrupt Enable
--- - Unimplemented, read as 0
CCP2IE - CCP2 Interrupt Enable

INTERRUPTS

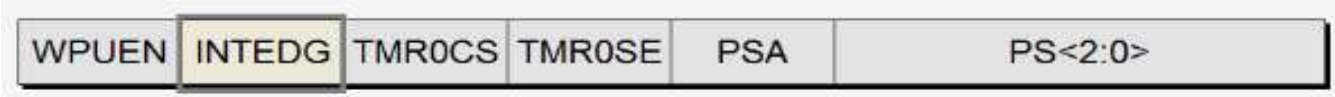
PIE3 register

----	CCP5IE	CCP4IE	CCP3IE	TMR6IE	----	TMR4IE	----
bit 7							bit 0

CC[illegible]
CCP4IE - CCP4 Interrupt Enable
CCP3IE - CCP3 Interrupt Enable
TMR6IE - Timer6 Interrupt Enable
--- - Unimplemented, read as 0
TMR4IE - Timer4 Interrupt Enable
--- - Unimplemented, read as 0

INTERRUPTS

OPTION_REG



The INTEDG flag in OPTION_REG is used to set a rising or failing edge on the INT pin as the trigger for an INTE interrupt.

VIDEOS

Web link

https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8bit-pic/peripherals/interrupts/