Interrupts on PIC micro

1. Lesson1 - introduction

If you are becoming a pic-microcontroller user and all new things are puzzling you, starting with interrupts would be "the hell thing" because you'll have to manage with your own imagination all pic resources, to got the final goal, a functional and stabile circuit board. Usually all advices you'll heard from older pic-micro users would be: "can't you do it without interrupts ? " or " the interrupts are difficult for beginners ! ". If you are reading now this issue, it's supposed you have taken already a carefull look on your favorite pic-micro data sheet . If not, you are encourage to do it, because everything you need to know is there. I will try just to clarify some interrupt aspects for your eyes (and mine too…) in six practical lessons. It's also supposed, you are familiar with JAL language for PIC-microcontrollers, you have already at least a blinking led running, resulted as your own program writing, compiling and PIC programming or much better, you have already connected your PIC with your own PC and talking between those without major breakdowns on your RS232 interface… If you don't know nothing about all these, you may got some information browsing here:

- http://www.voti.nl/jal/
- http://www.geocities.com/vsurducan/electro/PIC/pic.htm
- http://groups.yahoo.com/group/jallist/files/

Your first rightful ask would be: "Why we need interrupts?". The answer it's easy: because the PIC microcontroller it's stupid. "Should I choose then another microcontroller ? " No, because all microcontrollers are stupidus ! Only the user it's clever. PIC microcontroller is doing (almost like your PC) only one thing once. Don't try to ask him to solve two different problems in the same time, will not be possible. For this purpose you need a multi-PIC scheme. But PIC microcontrollers can stop it's own principal program execution (called **main** or **main loop** because here are chained all instructions which will run almost all of your tasks) and run a small program branch (called **interrupt service routine** because is launched by an **interrupt event** and usually it has a shorter length and is doing only a few things, opposite to main which have a considerable length). Because for hobbists and beginners, only flash midrange PIC-micro are user-friendly usable, having the greatest feature of being reprogramable [while are running: PIC16F87xA, PIC16F87x, or not: PIC16F7x, PIC16F62x, PIC16F8x, PIC12FXXX], I will talk only about interrupts on those. For these microcontrollers, the interrupt service routine will began always from the same address called **interrupt vector address** (address 04). Fortunately (or not?) for the user, can be only one **I**nterrupt **S**ervice **R**outine, even the program running inside the ISR may be splitted in many procedures.

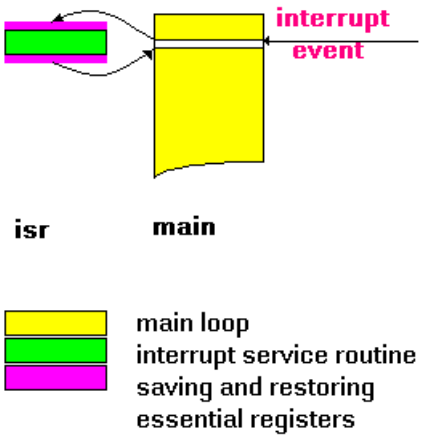

Figure 0-1 The interrupt mechanism

In JAL, the interrupt service routine must be written inside a procedure's body, the name of the procedure it's irrelevant because **will not be called** by another main procedures. The only reason why this procedure is necessary is to keep the **pragma interrupt** instruction. When the compiler find this instruction will put the assembled code written after this pragma at address 04. That's the replacement for **org 04h** directive in assembler to put the code (or the call instruction) at the interrupt vector address.

The problems are just begans. As you may know there are three extremely important registers (between the other 54 different registers in PIC16F877 or just 28 different registers in PIC12F675) : **w** register (the accumulator's result storage register), **status** register (containing flags of mathematic operations and bank switching bits) and the **FSR** register (containing 8 address bits for indirect addressing). If you have downloaded at least the JAL04.50 compiler all things are clear if you have inspected the assembler code resulted after a program compilation . If not, do it now after you'll compile the following jal lines:

```
include 16f84_4
include jpic

procedure whats_new_Stan is
 pragma interrupt
end procedure
```

The essential compilation result would be the next one, which have the necessary translation for used registers names in the right colum:

```
ORG 0000
  goto    __main
 ORG 0004                    ; SAVING ESSENTIAL REGISTERS (PUSH)
  movwf   H'0C'              ; movwf temporary_w
  swapf   H'03',w            ; swapf  status,w
  clrf    H'03'              ; clrf  status
  movwf   H'0D'              ; movwf temporary_status
  movfw   H'0A'              ; movfw pclath
  movwf   H'0E'              ; movwf temporary_pclath
  clrf    H'0A'              ; clrf  pclath
  movfw   H'04'              ; movfw FSR
  movwf   H'0F'              ; movwf temporary_FSR
  goto    __interrupt
 ORG 000E
__interrupt: ; 000E          ; interrupt user code
_7577__vector: ; 000E
p_7577_whats_new_stan: ; 000E
e_7577_whats_new_stan: ; 000E
                            ; RESTORING THE ESSENTIAL REGISTERS (POP)
  movfw   H'0F'              ; movfw temporary_FSR
  movwf   H'04'              ; movwf FSR
  movfw   H'0E'              ; movfw temporary_pclath
  movwf   H'0A'              ; movwf pclath
  swapf   H'0D',w            ; swapf temporary_status, w
  movwf   H'03'              ; movwf status
  swapf   H'0C',f            ; swapf temporary_w, f
  swapf   H'0C',w            ; swapf temporary_w, w
  retfie                     ; retfie
__main: ; 0017
```

The first observation it's the abuse of usage for *swapf* instruction. Jumping to the instruction set description of any PIC microcontroller, *swapf* shows there are no status flags affected by this instruction. The reason why it's used here, is because is not modifying the status register while saving and restoring the

other essential registers. For PIC microcontrollers having less than 2Kbytes memory pages (like PIC12F629/675, PIC16F84 or PIC16F62x) saving and restoring pclath it's not necessary. That's because the **P**rogram **C**ounter in PIC arhitecture has 13 bits wide, splitted in **P**rogram **C**ounter **L**ow and **P**rogram **C**outer **H**igh. Only PCL its readable and writable. The PCH it's writable only via PCLATH register. Any *call* or *goto* instructions provide only 11 bits of address and allow branching within any 2K program memory page. The upper two bits of the address are provided by PCLATH bites 4 and 3. Those two are essential in PICs having 4K program memory (PIC16F873/874 PIC16F73/74) or 8K program memory (PIC16F876/877 PIC16F76/77) because are addressing the desired program memory page. However the PC is pushed onto the 8 level stack every time a *call* instruction is executed or an interrupt causes a branch, but PCLATH is not affected by a push or a pop operation. Note, there is no physical *push* or *pop* instructions, but a sequence of saving-restoring which is used any time an interrupt occure, as the last assembler code shows. Another important thing is the *retfie* instruction at the end of the pop sequence which **allways** enables all unmasked interrupts by seting the *intcon_gie* flag. So the interrupts are native enabled in PIC micro arhitecture, **only after the first interrupt** has ocured and the ISR was readed. It's obviously clear you need to enable the interrupts for the first time somewhere inside the main; (the **main** program it contains a definition sequence (includes, register definitions) and the **main loop** *(forever loop ... end loop* sequence)), else your interrupt routine will not work at all. The ISR can be placed anywhere in the main, except in the **main loop**, and only after the registers used inside the ISR are defined.

Let's see which are the most important registers dealing with interrupts in PICmicro midrange arhitecture, first one is the 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	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **PEIE**: Peripheral Interrupt Enable bit
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT 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 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT 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; a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared (must be cleared in software).
0 = None of the RB7:RB4 pins have changed state

Figure 0-2 INTCON register

The *intcon_gie* flag must be enabled any time when we are reading an interrupt event and be disabled when the interrupts are interfering with our program branch. One example is writing a long message on the 4 rows, 20 characters LCD when the button read interrupts are enabled. The effect will be a stop in LCD writing when the reading button interrupt sequence its reached (as will see in Lesson2 Disabling interrupts. When?). The writing will continue after one button it's pushed. The cause of this malfunction is the time length necessary for LCD writing which is longer than button reading refresh rate. Solution: disable interrupts while the whole message are writted on the LCD or increase the refresh rate for buttons. Last choice is not applicable, because the refresh rate is usually set between 10mS and 60mS and writing on a lazy LCD can take much longer.

The *intcon_peie* flag may be used any time we are dealing with peripheral interrupts. The interrupt logic diagram of any PIC microcontroller will show us who are the peripheral interrupts:

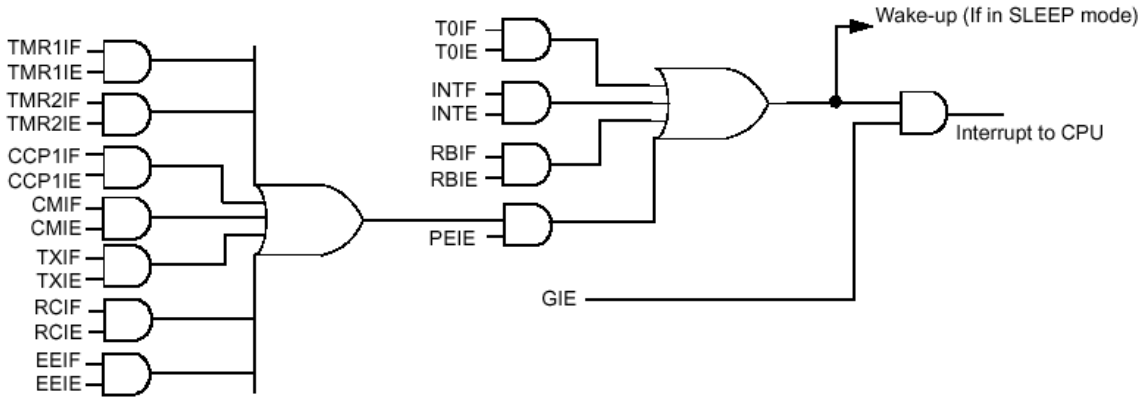

Figure 0-3 The interrupt logic in PIC16F62x microcontroller

The source of interrupts which can be masked by *intcon_peie* are dependent by microcontroller resources, in PIC16F628 they are:

- **TiMeR1 I**nterrupt,
- **TiMeR2 I**nterrupt,
- **C**ompare**C**apture**P**wm**1 I**nterrupt,
- **CoM**parator **I**nterrupt,
- **U**niversal **S**ynchronously **A**synchronously **R**eceiver **T**ransmiter **T**ransmit **I**nterrupt (TXI),
- **USART R**eceive **I**nterrupt (RXI),
- **EE**prom write operation **I**nterrupt

PIC16F877A have the same peripheral interrupts like PIC16F628 and a few more:

- **A**nalogic **D**igital module **I**nterrupt
- **C**ompare **C**apture **P**wm**2 I**nterrupt
- **S**ynchronous **S**erial **P**ort **I**nterrupt
- **P**arallel **S**lave **P**ort read-write**I**nterrupt
- **B**us **C**o**L**ision **I**nterrupt

PIC12F675 have the same peripheral interrupts as PIC16F628 except the USART interrupts because there is no USART module in PIC12F675.

All those interrupt sources have two special bits: the **F**lag bit and the **E**nable bit. The enable bit allows the interrupt to be set and the flag bit allows to read the effect of the interrupt. The flag bit must usually be cleared in software. Except for the interrupts which can be found inside the *intcon* register, the peripheral interrupts are defined by PI**E** registers (for **E**nabling the interrupts) and PIR registers (for reading the

interrupts effect via **F**lags). For the PIC16F877 there are two PIR and two PIE registers which are described below:

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
0 = The A/D conversion is not complete

bit 5 **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer is full
0 = The USART receive buffer is empty

bit 4 **TXIF**: USART Transmit Interrupt Flag bit
1 = The USART transmit buffer is empty
0 = The USART transmit buffer is full

bit 3 **SSPIF**: Synchronous Serial Port (SSP) Interrupt Flag
1 = The SSP interrupt condition has occurred, and must be cleared in software before returning from the Interrupt Service Routine. The conditions that will set this bit are:
- SPI
  - A transmission/reception has taken place.
- I^2C Slave
  - A transmission/reception has taken place.
- I^2C Master
  - A transmission/reception has taken place.
  - The initiated START condition was completed by the SSP module.
  - The initiated STOP condition was completed by the SSP module.
  - The initiated Restart condition was completed by the SSP module.
  - The initiated Acknowledge condition was completed by the SSP module.
  - A START condition occurred while the SSP module was idle (Multi-Master system).
  - A STOP condition occurred while the SSP module was idle (Multi-Master system).

0 = No SSP interrupt condition has occurred.

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 = TMR1 register did not overflow

Note 1: PSPIF is reserved on PIC16F873/876 devices; always maintain this bit clear.

Figure 0-4 PIR1 register in PIC16F877

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(1):** Parallel Slave Port Read/Write Interrupt Enable bit
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 converter interrupt
0 = Disables the A/D converter 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**: Synchronous Serial Port Interrupt Enable bit
1 = Enables the SSP interrupt
0 = Disables the SSP 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

Note 1: PSPIE is reserved on PIC16F873/876 devices; always maintain this bit clear.

Figure 0-5 PIE1 register in PIC16F877

All peripheral interrupts bits are quite clear described in those PIE-PIR registers. The user must set first the enable bit coresponding to the interrupts which will be used and then to read the coresponding flag bit and detetect the interrupt event inside the ISR. The only problem which the user may have, would be to choose the right reading sequence without loosing any asynchronously event. This will not be easy for multiple interrupts events (or chained interrupts events) which must be detected. A good rule is to disable the specific interrupts, while inside the ISR that interrupt event is treated, else (especialy for fast events) the program may loop in a false forever loop (because of an interrupt event which is taking place inside the same interrupt branch while the last interrupt is treated)

Going back to *intcon* register, there were left also three interrupts sources:

- **Tmr0 I**nterrupt
- **INT** RB0/**INT** external Interrupt
- **RBI RB** Port Change **I**nterrupt

TMR0 interrupt is the most frequently used interrupt for small time events detection. TMR0 is an 8 bit wide register having also a prescaler setable with 8 different rates from 1:2 to 1:256. The prescaler must be understood as a divider register. The TMR0 register will be incremented one step, from the value which has been previously written, every time when the prescaler has overflow. When TMR0 has overflow the coresponding *intcon_t0if* flag will be set. Inspecting this flag, use the low to high transition of the flag in conjunction with other time dependent events (ie: reading buttons, increment a real time clock register) and reseting the flag, it's an user defineable problem.

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

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

bit 6 **Reserved**: Always maintain this bit clear

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

bit 4 **EEIF**: EEPROM Write Operation Interrupt Flag bit

1 = The write operation completed (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 has occurred in the SSP, when configured for I2C Master mode
0 = No bus collision has occurred

bit 2-1 **Unimplemented:** Read as '0'

bit 0 **CCP2IF**: CCP2 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

Figure 0-6 PIR2 register for PIC16F877

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

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

bit 6 **Reserved**: Always maintain this bit clear

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

bit 4 **EEIE**: EEPROM Write Operation Interrupt Enable

1 = Enable EE Write Interrupt
0 = Disable EE Write Interrupt

bit 3 **BCLIE**: Bus Collision Interrupt Enable

1 = Enable Bus Collision Interrupt
0 = Disable Bus Collision Interrupt

bit 2-1 **Unimplemented:** Read as '0'

bit 0 **CCP2IE**: CCP2 Interrupt Enable bit

1 = Enables the CCP2 interrupt
0 = Disables the CCP2 interrupt

Figure 0-7 PIE2 register for PIC16F877

The RB0/INT Interrupt and RB Port Change Interrupt are the microcontroller's open eyes to the external world. Even I've heard rumors at piclist about the non realiable RB Port Change Interrupts, every time I've used those interrupts I was satisfied. The purpose of interrupt on change are to check the changing state of the port B4…B7 for fast or slow events. There aren't good olnly for **changing state** detection so don't try to use them like using the RB0 interrupt because will not work. Don't try also to detect fast events using pure JAL and portB change interrupt because will probably don't work also. But in pure assembler under JAL will be fine.

The RB0 interrupt it's perfect for mains zero cross detection (like will see in Lesson 3) or other external commands (like wake up from sleep on various external events). Using an external logic, the RB0/INT may be multiplexed to as many inputs we need, using the same scheme presented in figure 0.3 for peripheral interrupts inside the PIC.

There are minor differences in naming the interrupt bits which belongs to the interrupt registers, in various PIC microcontrollers. But the bits function are almost the same. For example the intcon register from PIC12F675. You may notice only the GPIF and GPIE differences against RBIF and RBIE from the other midrange PIC microcontrollers.

I've heard about many programs claiming they can transform the PIC into a multitasking tool using the interrupts in a fancy way. I have doubts there is an universal solution for using interrupts. However, a good starting point for understanding interrupts will be also:

AN585 – A Real-Time Operating systems for PIC micro Microcontrollers
AN576 – Tehniques to Disable Global Interrupts
AN566 – Using the PortB Interrupt on Change as an External Interrupt
All this application notes can be found in Embedeed Control Handbook, vol1 edited by Microchip, or on http://www.microchip.com