EEE 204

Introduction to Interrupts

Asst. Prof. Dr. Seydi KAÇMAZ

Introduction to Interrupts

- An interrupt is an approach to dealing with external, asynchronous events by building hardware into the MCU that handles identifying and prioritizing events to be serviced by the CPU.
- The interrupt system uses the concept of a flag to notify the CPU that an external event on a peripheral has occurred and action is requested.
- This approach allows the CPU to continue its normal instruction execution and only act when a flag is seen.
- When a flag is observed, the CPU completes its current instruction and then executes a sequence of instructions (written by the programmers) that accomplishes the desired action for the peripheral.
- The code that is executed when an interrupt occurs is called an **Interrupt Service Routine (ISR) or interrupt handler.**

Interrupt Priority and Enabling

- Interrupts have a priority system that ranks each external peripheral from highest to lowest. This provides a means to handle multiple interrupts that occur at the same time and are simultaneously requesting service from the CPU.

- An MCU has three categories of interrupts (sorting from highest to lowest)
  - **System Resets;** They are the most critical because they cause the MCU to start its operation from the beginning. This includes putting all configuration registers at their default values, initializing the program counter, and entering the main program at its first instruction.
  - **Non-maskable Interrupts (NMIs);** They are the second highest priority interrupts and typically handle fault conditions on the MCU. Examples of non-maskable interrupts are memory access errors and oscillator faults. When maskable interrupt occur, it can be handled after executing the current instruction.
  - **Maskable Interrupts (MIs).** They are the third category of interrupts and are the interrupts that handle all of the common peripherals on an MCU (i.e., ports, timers, serial communication, ADC, and DACs). Maskable interrupts have both global and local interrupt enables. The GIE bit in the status register is used as the global enable for all maskable interrupts. When it is set (GIE = 1), then maskable interrupts are allowed. Upon reset GIE = 0, meaning no maskable interrupts are enabled. When non-maskable interrupts occur, the current instructions and status are stored in stack for the CPU to handle the interrupt.

Interrupt Priority and Enabling

Each peripheral system then has a local interrupt enable (IE) that is configured in the control/status registers within the memory map. The global and local interrupt enable bits can be thought of as gating switches that allow the peripheral's flag to be observed by the CPU when configured.

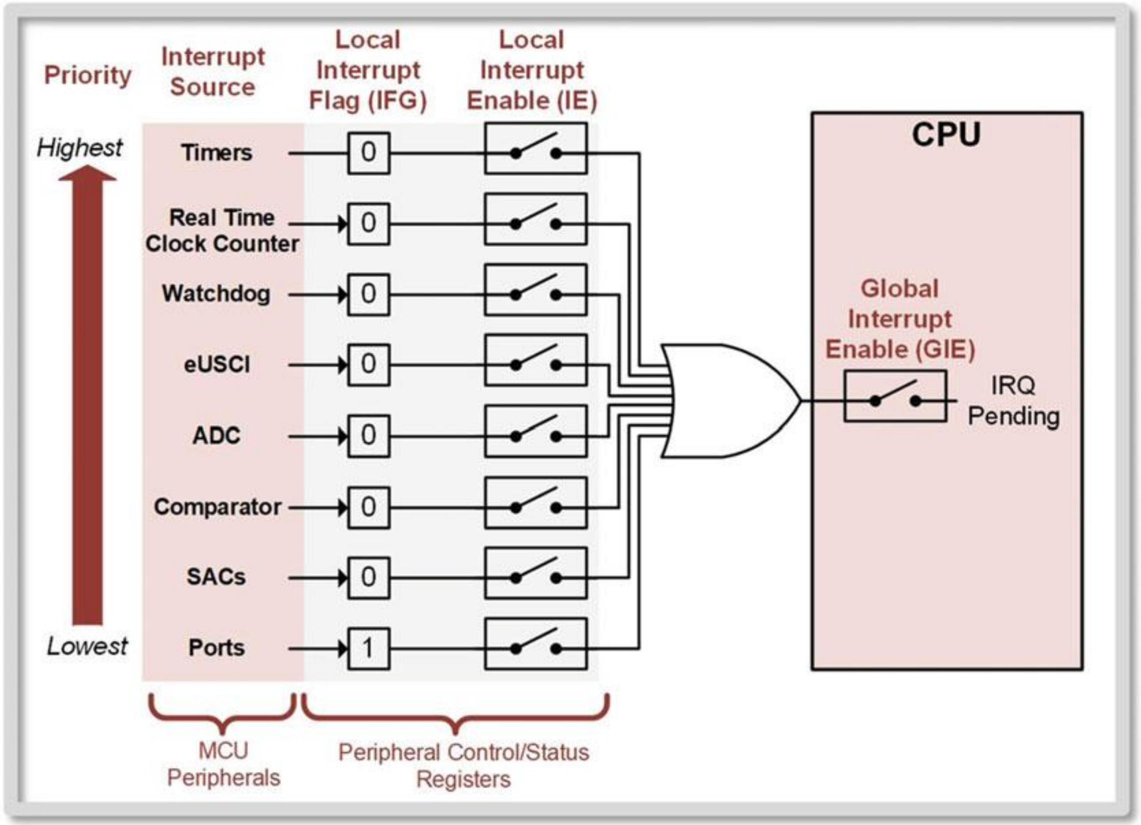


Fig. Conceptual model for global and local interrupt enables

Simply, If **Global Interrupt (GIE)** is **Enable**, **Local Interrupt (IE)** is **Enable** and **Local Interrupt Flag (IFG)** is Logic 1, Interrupt Service Routine is invoked.

While Global Interrupt Enable and Local Interrupt Enable are set manually in the program, Local Interrupt Flag becomes logic 1 if the correspondent event is realized. These events can be timer overflow/compare, port interrupts or any of them given in the **Interrupt Source** column given in the figure.

Interrupt Vectors

The beginning of an ISR is marked with an address label in the main.asm file. This address label serves as the starting address to be put into the PC when the ISR is called. The way that the CPU retrieves the starting address of the ISR to put into the program counter uses the concept of an ***interrupt vector.***

Each peripheral system that is capable of generating an interrupt is assigned a dedicated address location at the end of the program memory space. The address is called the peripheral's interrupt vector address and will hold the starting address of the ISR.

Since there are numerous peripherals that each require a unique vector, the addresses consume a block of memory called the ***interrupt vector table.***

The starting address of the ISR is put into the interrupt vector table when the program is downloaded.

Interrupt Service Routine (ISR) is a function that hardware invokes in response to an interrupt.

Interrupt Vectors

The full MSP430 architecture supports up to 64 separate interrupt vectors whose addresses are located within the range FF80H→FFFEH

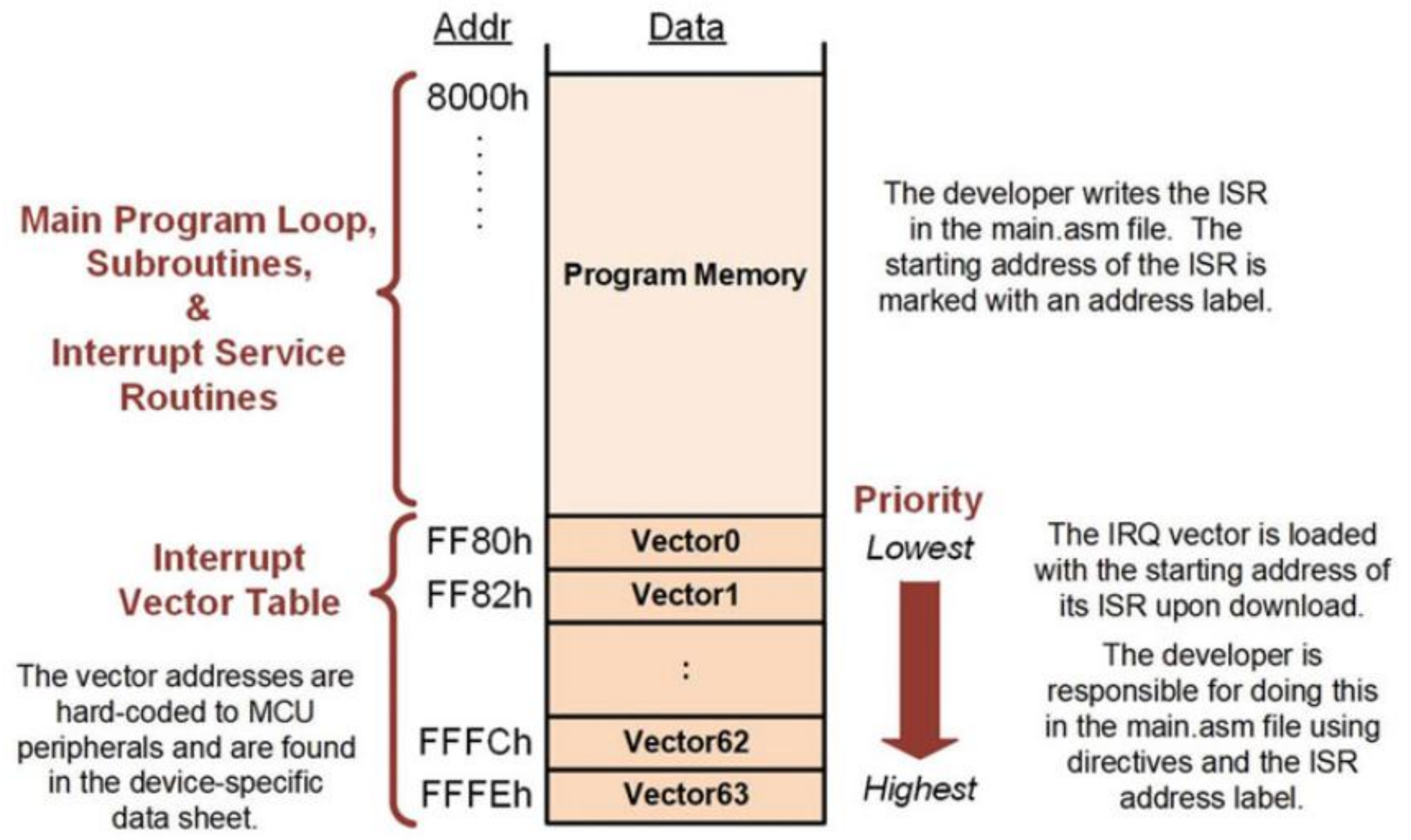
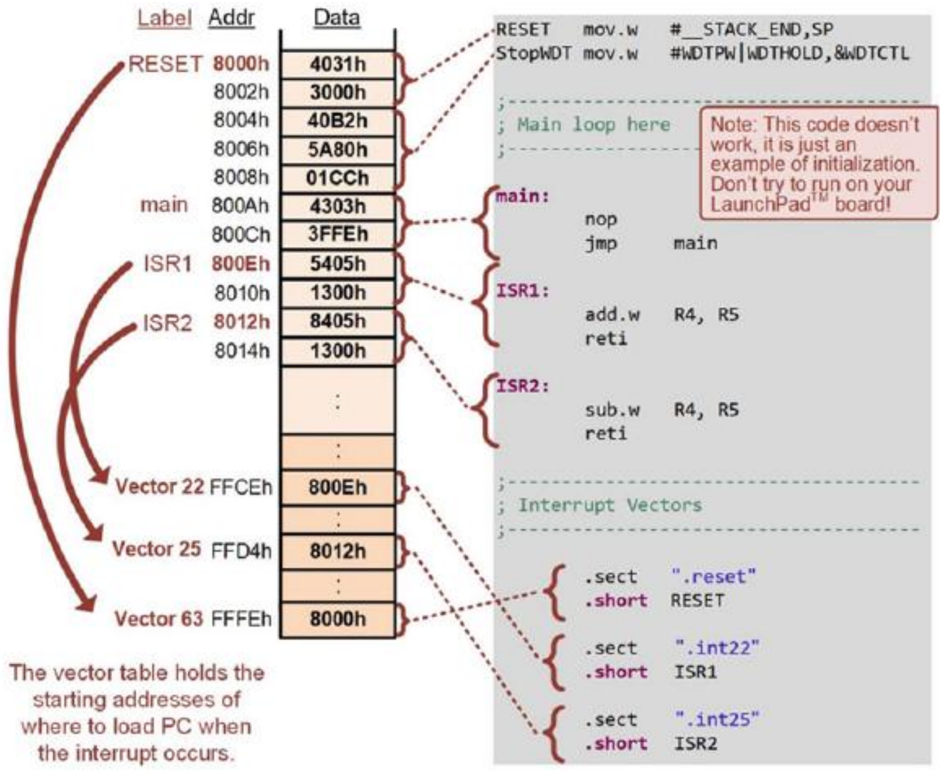

Addr
Data
8000h
Main Program Loop, Subroutines, & Interrupt Service Routines
Program Memory
The developer writes the ISR in the main.asm file. The starting address of the ISR is marked with an address label.
Interrupt Vector Table
FF80h
Vector0
FF82h
Vector1
FFFCh
Vector62
FFFEh
Vector63
The vector addresses are hard-coded to MCU peripherals and are found in the device-specific data sheet.
Priority
Lowest
Highest
The IRQ vector is loaded with the starting address of its ISR upon download.
The developer is responsible for doing this in the main.asm file using directives and the ISR address label.

Fig. Interrupt vector table concept

Interrupt Vectors

Figure shows a graphical depiction of how the interrupt vector table is initialized using the starting addresses of ISRs and assembler directives. In this figure, three interrupt vectors are initialized: reset, vector 22, and vector 25. Vectors 22 and 25 represent maskable interrupts that have ISRs that are executed when serviced. The starting addresses of these routines (named ISR1 and ISR2) are placed into their respective vector locations using the assembler directives `.section` and `.short`.

vector 63 handles the highest priority interrupt in the MCU, which is reset. This vector holds the starting address of where to begin executing code when the MCU is reset or powered up.



```
RESET   mov.w   #__STACK_END,SP
StopWDT mov.w   #WDTPW|WDTHOLD,&WDTCTL

;-------------------------------------------
; Main loop here
;-------------------------------------------
main:
        nop
        jmp     main

ISR1:
        add.w   R4, R5
        reti

ISR2:
        sub.w   R4, R5
        reti

;-------------------------------------------
; Interrupt Vectors
;-------------------------------------------
        .sect   ".reset"
        .short  RESET

        .sect   ".int22"
        .short  ISR1

        .sect   ".int25"
        .short  ISR2
```

Fig. A graphical depiction of initializing the interrupt vector table

Interrupt Service Routines

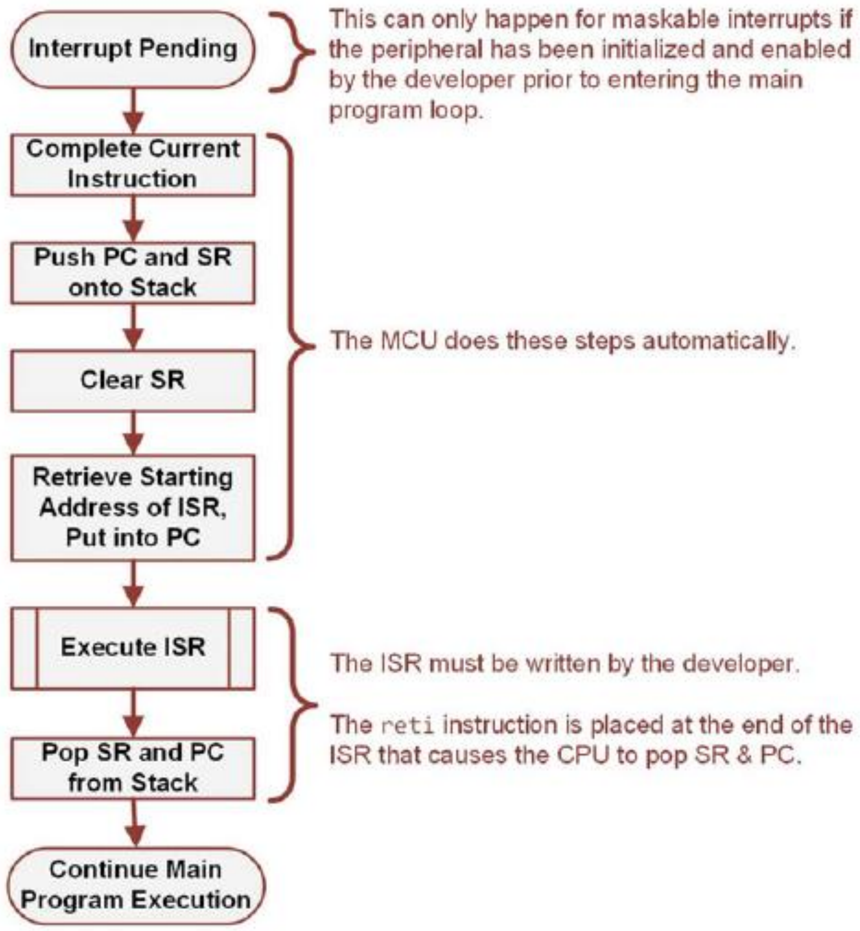
An interrupt service routine is written in a similar manner as a subroutine. They both need to start with an address label and contain instructions to be executed when called; however, an ISR must end with a dedicated instruction called ***return from interrupt (`reti`).*** The `reti` instruction will pop the SR and PC off of the stack in order to return the CPU execution back to the main program.

Another critical role of a maskable ISR is that it must clear the peripheral's local interrupt flag (IFG) that caused the interrupt in the first place. **If the IFG is not cleared, then as soon as the ISR completes and the CPU returns to the main program, it will be immediately interrupted again because the flag is still asserted. This leads to an infinite ISR loop that the CPU can never get out of.**

ISRs should be short, fast, and dedicated to only performing the functionality needed by the peripheral at that time. A good ISR should impact the rest of the MCU as little as possible.

Interrupt Servicing Summary

Figure shows a flow chart of the steps that are taken when an IRQ (Interrupt Request) is serviced. Note that some of the steps are taken automatically by the CPU while some are up to the developer.



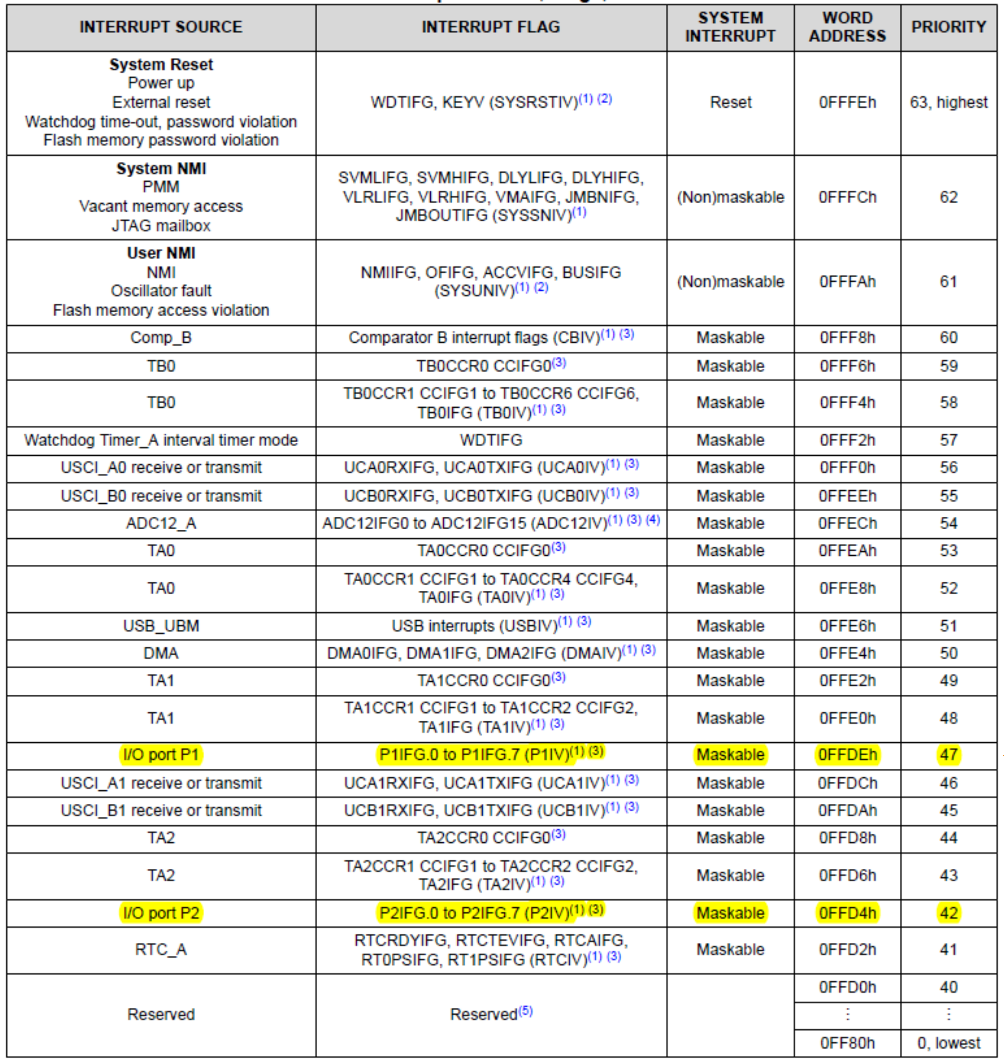
Interrupt Pending
This can only happen for maskable interrupts if the peripheral has been initialized and enabled by the developer prior to entering the main program loop.
Complete Current Instruction
Push PC and SR onto Stack
Clear SR
The MCU does these steps automatically.
Retrieve Starting Address of ISR, Put into PC
Execute ISR
The ISR must be written by the developer.
Pop SR and PC from Stack
The reti instruction is placed at the end of the ISR that causes the CPU to pop SR & PC.
Continue Main Program Execution

Fig. Sequence of tasks performed when servicing an interrupt

When using maskable interrupts, it is important to keep in mind which tasks are taken care of automatically by the MCU and those that the developer must do. When using a specific peripheral with a maskable interrupt, the developer has the following responsibilities:

1. Configure the peripheral for the desired functionality.
2. Clear the peripheral’s interrupt flag (IFG).
3. Assert the local interrupt enable (IE) for the peripheral.
4. Assert the global interrupt enable (GIE) in the status register.
5. Write the ISR with an address label to mark its starting location and the `reti` instruction to denote its end. **Remember that the ISR must clear the peripherals local interrupt flag (IFG) so that when the ISR completes, the peripheral doesn’t inadvertently trigger another IRQ.**
6. Initialize the vector address for the peripheral using the ISR address label and assembler directives.

Interrupt Servicing Summary



INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power up External reset Watchdog time-out, password violation Flash memory password violation	WDTIFG, KEYV (SYSRSTIV)(1) (2)	Reset	0FFFEh	63, highest
System NMI PMM Vacant memory access JTAG mailbox	SVMLIFG, SVMHIFG, DLYLIFG, DLYHIFG, VLRLIFG, VLRHIFG, VMAIFG, JMBNIFG, JMBOUTIFG (SYSSNIV)(1)	(Non)maskable	0FFFCh	62
User NMI NMI Oscillator fault Flash memory access violation	NMIIFG, OFIFG, ACCVIFG, BUSIFG (SYSUNIV)(1) (2)	(Non)maskable	0FFFAh	61
Comp_B	Comparator B interrupt flags (CBIV)(1) (3)	Maskable	0FFF8h	60
TB0	TB0CCR0 CCIFG0(3)	Maskable	0FFF6h	59
TB0	TB0CCR1 CCIFG1 to TB0CCR6 CCIFG6, TB0IFG (TB0IV)(1) (3)	Maskable	0FFF4h	58
Watchdog Timer_A interval timer mode	WDTIFG	Maskable	0FFF2h	57
USCI_A0 receive or transmit	UCA0RXIFG, UCA0TXIFG (UCA0IV)(1) (3)	Maskable	0FFF0h	56
USCI_B0 receive or transmit	UCB0RXIFG, UCB0TXIFG (UCB0IV)(1) (3)	Maskable	0FFEEh	55
ADC12_A	ADC12IFG0 to ADC12IFG15 (ADC12IV)(1) (3) (4)	Maskable	0FFECh	54
TA0	TA0CCR0 CCIFG0(3)	Maskable	0FFEAh	53
TA0	TA0CCR1 CCIFG1 to TA0CCR4 CCIFG4, TA0IFG (TA0IV)(1) (3)	Maskable	0FFE8h	52
USB_UBM	USB interrupts (USBIV)(1) (3)	Maskable	0FFE6h	51
DMA	DMA0IFG, DMA1IFG, DMA2IFG (DMAIV)(1) (3)	Maskable	0FFE4h	50
TA1	TA1CCR0 CCIFG0(3)	Maskable	0FFE2h	49
TA1	TA1CCR1 CCIFG1 to TA1CCR2 CCIFG2, TA1IFG (TA1IV)(1) (3)	Maskable	0FFE0h	48
I/O port P1	P1IFG.0 to P1IFG.7 (P1IV)(1) (3)	Maskable	0FFDEh	47
USCI_A1 receive or transmit	UCA1RXIFG, UCA1TXIFG (UCA1IV)(1) (3)	Maskable	0FFDCh	46
USCI_B1 receive or transmit	UCB1RXIFG, UCB1TXIFG (UCB1IV)(1) (3)	Maskable	0FFDAh	45
TA2	TA2CCR0 CCIFG0(3)	Maskable	0FFD8h	44
TA2	TA2CCR1 CCIFG1 to TA2CCR2 CCIFG2, TA2IFG (TA2IV)(1) (3)	Maskable	0FFD6h	43
I/O port P2	P2IFG.0 to P2IFG.7 (P2IV)(1) (3)	Maskable	0FFD4h	42
RTC_A	RTCRDYIFG, RTCTEVIFG, RTCAIFG, RT0PSIFG, RT1PSIFG (RTCIV)(1) (3)	Maskable	0FFD2h	41
Reserved	Reserved(5)		0FFD0h	40
			⋮	⋮
			0FF80h	0, lowest

System Reset

Non-maskable Interrupts

Maskable Interrupts

Port Interrupts

Fig. MSP430F5529 interrupt vector table (for assembly)

MSP430F5529 Port Interrupts

The port interrupt system provides a prioritization scheme that can speed up determining which bit of the port should be serviced first if multiple IRQs occur on a port simultaneously. The port interrupt system prioritizes bit 0 as the highest priority and bit 7 as the lowest priority within the port. Dedicated registers called the Port Px Interrupt Vector Word (PxIV, or P1IV and P2IV) registers are used to indicate priority when simultaneous port IRQs have occurred.

PxIV is loaded with a unique number corresponding to the bit that has just triggered an IRQ and also has the highest priority of any bits within the port that may have also triggered an IRQ. This number can be used within the service routine to quickly jump to the code associated with the highest priority bit. PxIV does not have a bitwise correspondence to the port bit that caused the interrupt. The values it takes on represent which of the 8 inputs is pending with the highest priority. The values it will take on are: bit0 = 02h, bit1 = 04h, bit2 = 06h, bit3 = 08h, bit4 = 0Ah, bit5 = 0Ch, bit6 = 0Ch, and bit7 = 10h.

MSP430F5529 Port Interrupts

Each bit within ports 1 and 2 on the MSP430F5529 has the ability to trigger an interrupt when configured as an input and there is a transition on its pin. Ports 1 and 2 each have their own dedicated vector address; however, each bit within each port all share the port's vector. So, it is the job of the developer to determine which bit triggered the interrupt manually and which part of the associated ISR to execute to service that bit.

The local enable for the port interrupts are configured in the Port Px Interrupt Enable (PxIE, or P1IE, and P2IE) registers. Each bit within PxIE corresponds to the bit of the port (i.e., bit 0 of P1IE enables the interrupt on bit 0 of P1). **A 0 in PxIE indicates that the interrupt for that bit of the port is disabled. A 1 in PxIE indicates that the interrupt for that bit of the port is enabled.** PxIE is cleared on reset, disabling the port interrupts. Since the port interrupts are maskable, the global interrupt for all bits is the GIE bit in the status register.

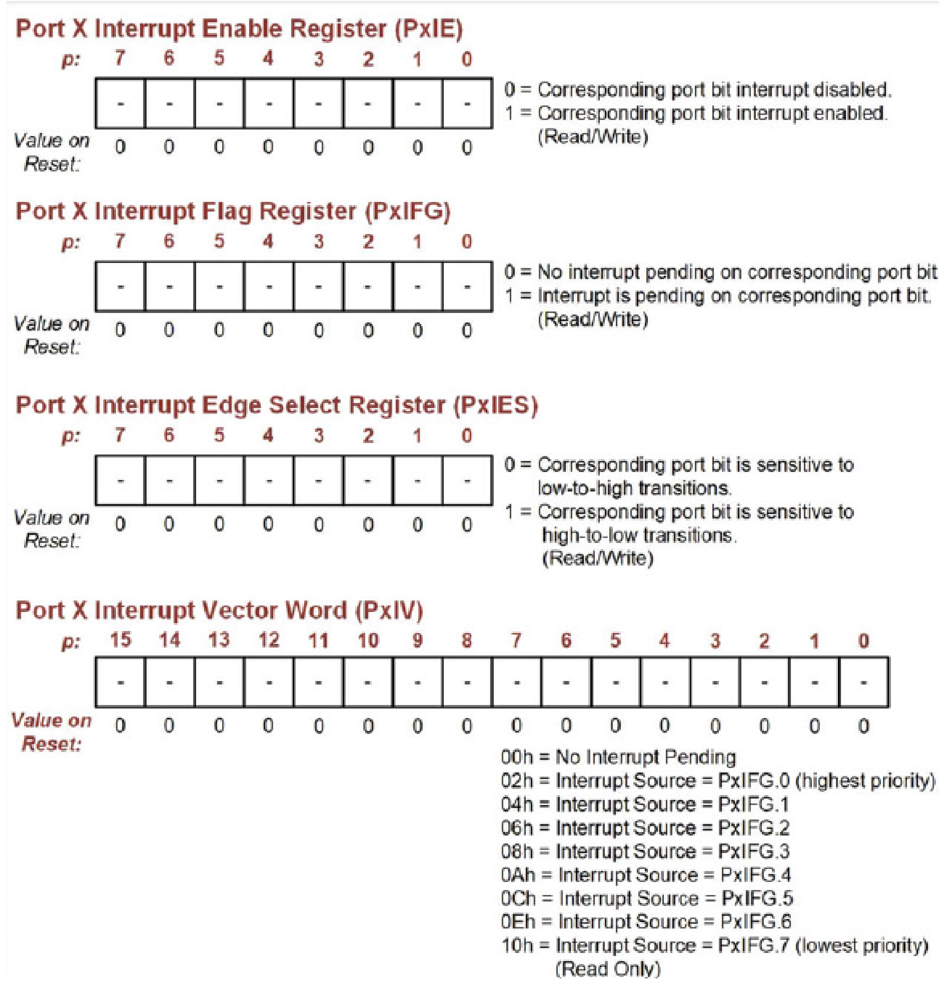
The flags for the port interrupts are held in the Port Px Interrupt Flag (PxIFG, or P1IFG and P2IFG) registers. Upon reset, all bits in PxIFG are set to 0. Upon an interrupt, the flag is asserted. Each bit within PxIFG corresponds to the bit of the port (i.e., if bit 0 of P1IFG is asserted it means an interrupt has occurred on bit 0 of P1). **Once a port IRQ is serviced, the bit's flag needs to be cleared in PxIFG by the developer.**

MSP430F5529 Port Interrupts

The last configuration setting for port interrupts is the ability to select which transition polarity triggers the interrupt (i.e., rising or falling). The Port Px Interrupt Edge Select (PxIES or P1IES and P2IES) registers. **A 0 in this register means the IRQ will be triggered on a low-to-high transition on the pin. A 1 in this register means the IRQ will be triggered on a high-to-low transition.** Each bit within this register corresponds to the bit in the port it configures (i.e., if bit 0 of P1IES is a 1, then a high-to-low transition on bit 0 of P1 will trigger an IRQ).

MSP430F5529 Port Interrupts

Figure gives a summary of the port interrupt configuration registers.



Port X Interrupt Enable Register (PxIE)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = Corresponding port bit interrupt disabled.
1 = Corresponding port bit interrupt enabled.
(Read/Write)

Port X Interrupt Flag Register (PxIFG)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = No interrupt pending on corresponding port bit
1 = Interrupt is pending on corresponding port bit.
(Read/Write)

Port X Interrupt Edge Select Register (PxIES)

p:	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0

0 = Corresponding port bit is sensitive to low-to-high transitions.
1 = Corresponding port bit is sensitive to high-to-low transitions.
(Read/Write)

Port X Interrupt Vector Word (PxIV)

p:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Value on Reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

00h = No Interrupt Pending
02h = Interrupt Source = PxIFG.0 (highest priority)
04h = Interrupt Source = PxIFG.1
06h = Interrupt Source = PxIFG.2
08h = Interrupt Source = PxIFG.3
0Ah = Interrupt Source = PxIFG.4
0Ch = Interrupt Source = PxIFG.5
0Eh = Interrupt Source = PxIFG.6
10h = Interrupt Source = PxIFG.7 (lowest priority)
(Read Only)

Fig. Summary of port interrupt configuration registers

MSP430F5529 Port Interrupts

When using port interrupts, there is a recommended initialization sequence to avoid inadvertent bit assertions of flags due to the nature of power on. The recommended sequence from the MSP430F5529 data sheet to configure a port interrupt is as follows:

1. Initialize the port direction (PxDIR), pull-up/down resistor (PxREN), the pull-up/down resistor polarity (PxOUT), and the port interrupt edge select (PxIES).

2. Clear the port interrupt flags (PxIFG) for first use. Note that the reset value for PxIFG=00h, but often bits will be asserted inadvertently due to step 1.

3. Assert the local port interrupt enable (PxIE).

4. Assert the global enable for maskable interrupts (GIE bit in SR).

MSP430F5529 Port Interrupts, Example:

Let’s now look at configuring the push-button switch S1 on the LaunchPad board to trigger a port interrupt. Let’s design a program that will toggle LED1 each time S1 is pressed using an interrupt. First, let’s look at the signal behavior of S1 when pressed. Figure shows a graphical depiction of the logic levels and transitions that occur when S1 is pressed and released.

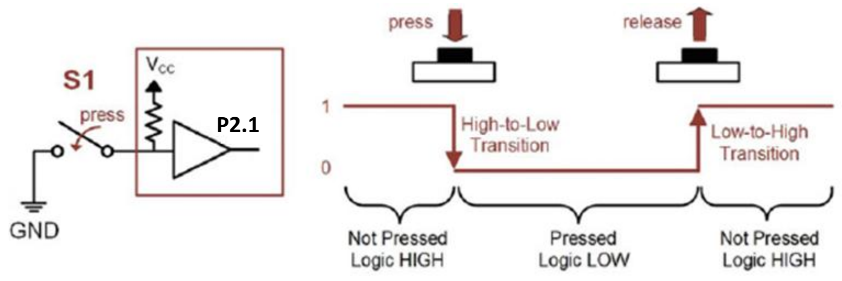


Fig. Signal behaviour of P2.1 when S1 is pressed and released

A few things to keep in mind when setting up a push-button interrupt:

• S1 is connected to Port2, bit 1. While the logic level of S1 can be observed on P2.1, when using an interrupt, we don’t have to look at this bit. We instead allow a transition to assert an interrupt flag and have the CPU execute an ISR accordingly.

• S1 is a switch that is connected to Vcc. This means we don’t need a pull-up resistor on the MCU to provide the logic HIGH state when S1 is not pressed. If we want the IRQ to trigger immediately upon a button press, then we need to configure the interrupt edge sensitivity (P2IES.1) to be high-to-low. When S1 is not pressed, P2.1 is at a logic high. When the button is pressed, P2.1 goes to a logic low. If we leave the P2IES.1 sensitivity at its default value of low-to-high sensitivity, the interrupt will only trigger once S1 is released.

• Since we are only using one bit within P2 to trigger an IRQ, we don’t need to use the P2IV register to determine the highest priority bit that caused the IRQ. We will simply use the P2IFG register knowing that we only care about bit1.

• The port 2 interrupt vector address is FFD4h. This has a CCS section name of .int42. This is the name we will use when we initialize the vector address using assembler directives.

• In the ISR, we will need to toggle LED1, clear the P2IFG.1 flag, and use `reti` to return from the interrupt.

MSP430F5529 Port Interrupts, example

```
init:
bis.b #BIT0, &P1DIR
bic.b #BIT0, &P1OUT
```

Set LED1 to be output (P1DIR.0=1)
Clear LED 1 initial (P1OUT.0=0)

```
bic.b #BIT1, &P2DIR
bis.b #BIT1, &P2REN
bis.b #BIT1, &P2OUT
bis.b #BIT1, &P2IES

bic.b #BIT1, &P2IFG
bis.b #BIT1, &P2IE

bis.w #GIE,  SR
```

Setup S1 as a port interrupt
- Set port direction to input (P2DIR.1=0)
- Enable Pull-Up/Down Resistor (P2REN.1=1, not needed for LaunchPad)
- Configure resistor as pull-up (P2OUT.1=1 not needed for LaunchPad)
- Set IRQ sensitivity High-to-Low (P2IES.1=1), means to toggle when button is **pressed**
- Clear Interrupt flag (P2IFG.1=0)
- Assert Local enable (P2IE.1=1)
- Assert Global enable (GIE=1,)

```
main:
jmp main
```

Main program, doesn't do anything but loop forever. However, it can also do something(s) and they must be before `jmp main`

```
ISR_S1:
xor.b #BIT0, &P1OUT
bic.b #BIT1, &P2IFG
reti
```

ISR toggles LED1 and clears the P2IFG.1 flag to return to normal program flow

```
; Stack Pointer definition
            .global __STACK_END
            .sect   .stack
```

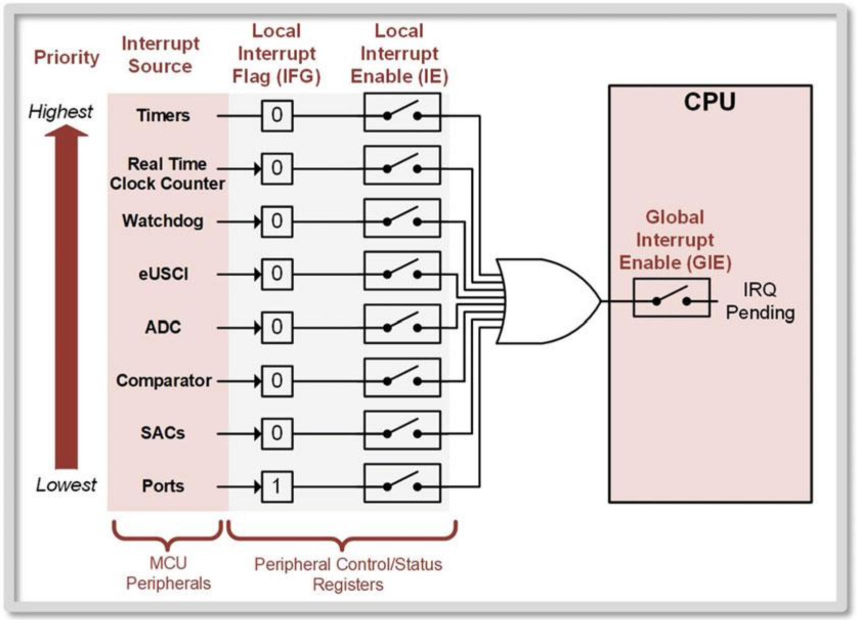
Written by the CCS compiler

```
; Interrupt Vectors
            .sect   ".reset"
            .short  RESET
```

Reset Vector and Written by the CCS compiler

```
            .sect   ".int42"
            .short  ISR_S1
```

Our Port2 interrupt is `.int42`. For Port1, `.int47` must be used



Remember this!