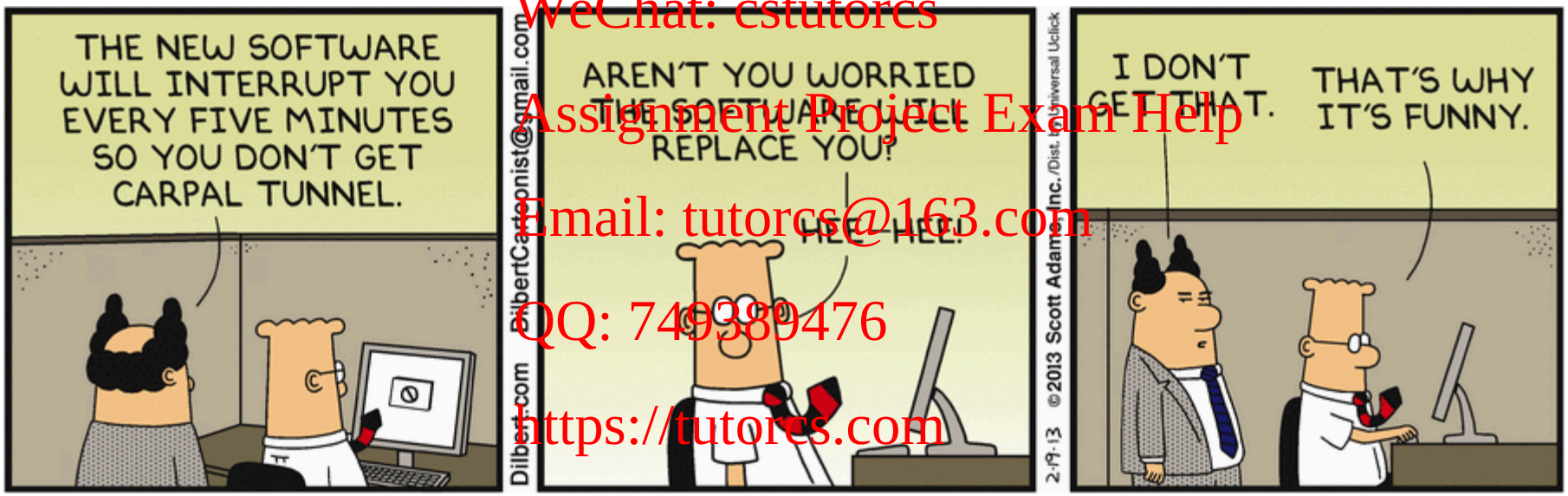
cture 21

rrupts



Joke of the Day

Why do programmers always mix up Halloween and Christmas?

Because Oct 31 = Dec 25

$(31)_8 = (25)_{10}$

Quiz 6 Posted – Due Wd April 5

ECE 2560 Intro to Microcontroller-Based Systems Spring 2023

uiz #6

Due: Wedl 5 – by 4:10 PM (Before Class)
Submission: Media Recording and File Upload to Carmen

This quiz is individual work. You can consult class materials, you can research algorithms (books, www etc.) but you are not allowed to collaborate with others.

Part 1: Coding Task (50 pts)

Your program should start with both LEDs off (i.e., not emitting light) and wait for a push button to be pressed. When either push button is pressed, an interrupt should be triggered on the raising edge. A single interrupt routine serves the interrupts and accomplishes following task:

- Pressing S1 toggles the **green LED**
- Pressing S2 toggles the **red LED**

Toggling an LED means the following: if the LED is off, it is turned on; alternatively, if the LED is on, it is turned off.

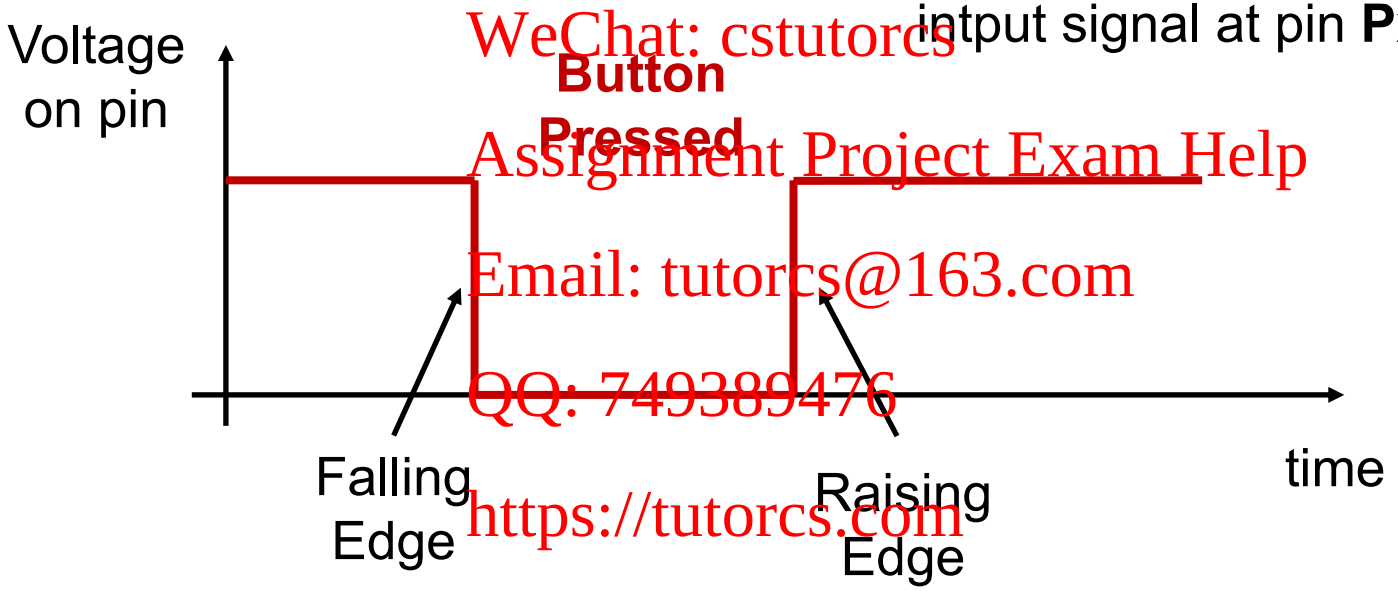
Your program should let you press the buttons as many times as you want, and in any order, and exhibit correct behavior.

Recap: Active Low Buttons

The push buttons S1 and S2 (and reset switch S3) are **active low buttons**

- when the switch is pressed they send a LOW or “0” signal
- when the switch is open they send a HIGH or “1” signal

PxIN.y reflects the value of the intput signal at pin **Px.y**



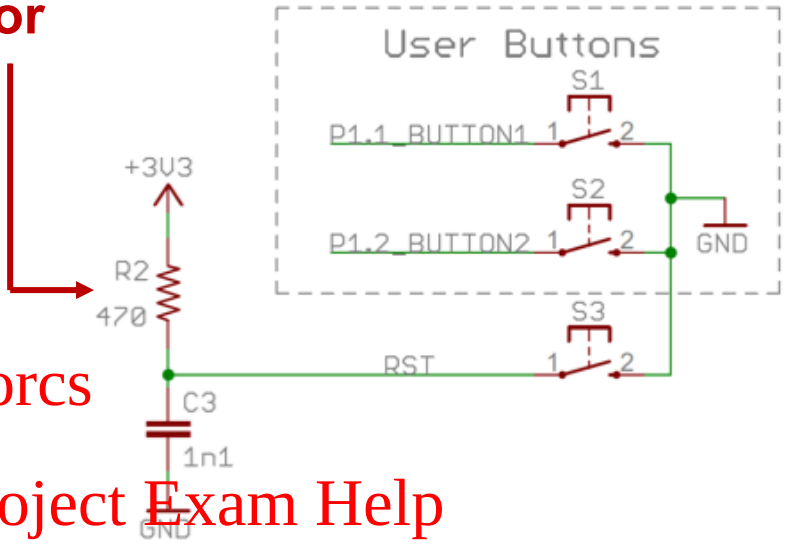
Spoiler: we will select falling or raising edge to trigger interrupts

Recap: Configuring the Resistor

Active low buttons require a **pullup resistor**

Pullup or Pulldown Res

Enable Register: PxREN



PxREN.y = 1: Resistor enabled

Output Register: PxOUT (Role 2)

Bit **PxOUT.y** selects pullup or pulldown at pin **Px.y**

PxOUT.y = 1: Pin **Px.y** is pulled up

if the pin is configured as I/O function, **input** direction and the pullup or pulldown resistor are enabled

Recap: Reading the Input at Pin Px.y

Input Register: PxIN

Bit **PxIN.y** reflects the val[illegible]ntput signal at pin **Px.y**

PxIN.y = 0: Inpu[illegible]**.y** is LOW

PxIN.y = 1: Input at pin **Px.y** is HIGH

How can we read the value? **When do we read the value?**

We will use the push buttons to trigger interrupts!!

There are three more port registers to configure for interrupts

- PxIE – Interrupt Enable
- PxIES – Interrupt Edge Select
- PxIFG Interrupt Flag

Configuring Px.y for Output

Only output we will use are the **red LED** and **green LED**

.0 **P9.7**

Step by Step Configura utput:

(**0.** Select pin functionality: PxSEL0.y = 0 and PxSEL1.y = 0) default

1. **Set desired output value:**

   PxOUT.y = 0 (LED off) or

   PxOUT.y = 1 (LED on)

Order of configuration matters: Otherwise, the initial output may be random

2. **Set direction to output: PxDIR.y = 1**

3. **Clear LOCKLPM5** – otherwise GPIO is not powered

```
bic.w #LOCKLPM5, &PM5CTL0
```

Configuring Px.y for Input

Only input we will use are **Push Button S1** and **Push Button S2**

P1.1 **P1.2**

Step by Step Configura put:

(0. Select Pin Functionality: PxSEL0.y = 0 and PxSEL1.y = 0) default

(1. Set direction to input: PxDIR.y = 0) default

2. **Enable resistor: PxREN.y = 1**
3. **Pullup resistor: PxOUT.y = 1** (2nd Role of PxOUT)
4. **Select interrupt triggering edge:**

   PxIES.y = 0 Interrupt triggered on raising-edge

   PxIES.y = 1 Interrupt triggered on falling-edge
5. **Enable interrupts for pin: PxIE.y = 1**
6. **Enable general interrupts in Status Register: `eint`**

Interrupts

First we had only the **main program**:

- PC points to next instruction to be executed
- CPU fetches, decodes, executes the instruction
- PC proceeds to the next instruction
- CPU fetches, decodes, executes the instruction …

Even with **subroutines** the picture did not change much:

- With a subroutine call, the old PC is saved on stack
- PC points to next instruction, CPU executes the instruction …
- When the instruction is a return the old PC is recovered from stack
- PC points to next instruction, CPU executes the instruction …

In many real-life applications, the CPU is idle for most of the time

– e.g., the infinite loop at the end of our programs

– better, in a low power mode, **waiting for external input**

How does the CPU know there is an input? **via an interrupt**

Interrupts

Interrupts are events that temporarily suspend this cycle of execution:

A peripheral (GPIO, eUSCI, etc.) can raise a **flag** indicating that it needs immediate attention (button, incoming data, expired timer etc.)

⇒ **Interrupt Request (IRQ)**

e.g., Push buttons S1 and S2 raise a flag in the **P1IFG** register

CPU completes executing the current instruction (up to 5 cycles!)

and starts *servicing* the interrupt by running a special *routine*

⇒ **Interrupt Service Routine (ISR) or Interrupt Handler**

Similar to a subroutine but with no external call – *starts running on its own*

MSP430 uses **vectored interrupts** – the address of the ISR that will be run when a certain IRQ is raised is stored in a vector table

⇒ **Interrupt Vector Table (IVT)**

A Special Interrupt

Every program we have written so far had interrupt handling: **RESET**

ISR – initialize st*pty*, PC points to the top of instructions

```
;
RESET       mov.w   #__STACK_E          ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDT          ; Stop watchdog timer


;-------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------


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

;-------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------
            .sect   ".reset"            ; MSP430 RESET Vector
            .short  RESET
```


Flag

Interrupt Vector Table

Corresponding ISR

We will add more interrupt handlers to this list

Different Types of Interrupts

In the MSP430 (and most other MCU too) here are three types of interrupts:

- System Reset
- Non-maskable Interrup
- Maskable Interrupt

Highest priority to lowest

Interrupts have a **priority system**: when multiple IRQs are issued simultaneously, the CPU executes the ISR for the interrupt with the highest priority

Check out **Table 6-4. Interrupt Sources, Flags, and Vectors** in the data sheet for MSP430FR6869

https://www.ti.com/lit/ds/symlink/msp430fr6989.pdf

Data sheet available on Carmen **slas789d.pdf**

Interrupts in MSP430FR6989

Table 6-4. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
System Reset Power up, Brownout, Supply Supervisor External Reset RST Watchdog time-out (watchdog mode) WDT, FRCTL MPU, CS, PMM password violation FRAM uncorrectable bit error detection MPU segment violation FRAM access time error Software POR, BOR	IFG STIFG FG UPW, CSPW, PMMPW FG 1IFG, MPUSEG2IFG, MPUSEG3IFG ACCTEIFG PMMPORIFG, PMMBORIFG (SYSRSTIV) (1) (2)	Reset	0FFFEh	Highest
System NMI Vacant memory access JTAG mailbox FRAM bit error detection MPU segment violation	VMAIFG JMBINIFG, JMBOUTIFG CBDIFG, UBDIFG MPUSEGIIFG, MPUSEG1IFG, MPUSEG2IFG, MPUSEG3IFG (SYSSNIV) (1) (3)	(Non)maskable	0FFFCh	
User NMI External NMI Oscillator fault	NMIIFG, OFIFG (SYSUNIV) (1) (3)	(Non)maskable	0FFFAh	
Comparator_E	Comparator_E interrupt flags (CEIV) (1)	Maskable	0FFF8h	
Timer_B TB0	TB0CCR0.CCIFG	Maskable	0FFF6h	
Timer_B TB0	TB0CCR1.CCIFG to TB0CCR6.CCIFG, TB0CTL.TBIFG (TB0IV) (1)	Maskable	0FFF4h	
Watchdog timer (interval timer mode)	WDTIFG	Maskable	0FFF2h	
Extended Scan IF	ESIIFG0 to ESIIFG8 (ESIIV) (1)	Maskable	0FFF0h	
eUSCI_A0 receive or transmit	UCA0IFG: UCRXIFG, UCTXIFG (SPI mode) UCA0IFG:UCSTTIFG, UCTXCPTIFG, UCRXIFG, UCTXIFG (UART mode) (UCA0IV)(1)	Maskable	0FFEEh	

Interrupts in MSP430FR6989

Table 6-4. Interrupt Sources, Flags, and Vectors (continued)

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
eUSCI_B1 receive or transmit)	UC CTXIFG (SPI mode) UCB IACKIFG, UCSTTIFG, UCS CTXIFG0, UCRXIFG1, UCT CTXIFG2, UCRXIFG3, UCT CBIT9IFG (I^2C mode))(1)	Maskable	0FFE2h	
DMA	DMA0CTL.DMAIFG, DMA1CTL.DMAIFG, DMA2CTL.DMAIFG (DMAIV)(1)	Maskable	0FFE0h	
Timer_A TA1	TA1CCR0.CCIFG	Maskable	0FFDEh	
Timer_A TA1	TA1CCR1.CCIFG to TA1CCR2.CCIFG, TA1CTL.TAIFG (TA1IV)(1)	Maskable	0FFDCh	
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)	Maskable	0FFDAh	
Timer_A TA2	TA2CCR0.CCIFG	Maskable	0FFD8h	
Timer_A TA2	TA2CCR1.CCIFG TA2CTL.TAIFG (TA2IV)(1)	Maskable	0FFD6h	
I/O Port P2	P2IFG.0 to P2IFG.7 (P2IV)(1)	Maskable	0FFD4h	

Port P1 generates maskable interrupts

There is a **single interrupt flag register** for all 8 pins of P1: **P1IFG**

There is a **single interrupt vector** for P1 ⇒ **Single ISR** serves all interrupts raised by P1.0 … P1.7

Revisiting the Status Register SR/R2

The Status Register SR plays an important role in interrupt handling

The GIE bit in the status register determines whether maskable interrupts are

- enabled **GIE = 1**
- or disabled **GIE = 0**

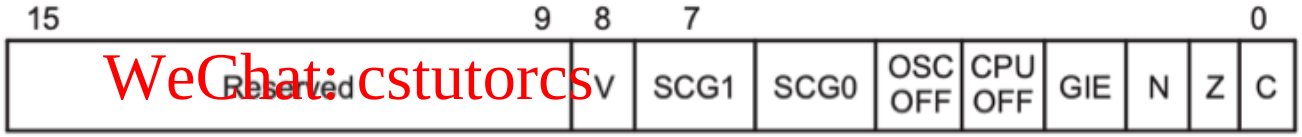
Several ways to set/clear this bit – easiest is to use dedicated instructions

```
eint  ; enable general interrupts
dint  ; disable general interrupts
```

Interrupt Handling

What happens when an interrupt flag is raised?

- CPU completes execu rent instruction
- Program Counter **PC** i onto the stack
- Status Register **SR** is to the stack
- SR is cleared



Low power modes disabled

Further interrupts disabled

- The highest priority interrupt is selected …
- ... its **ISR** is identified from the **IVT** …
- … address of the **ISR** is loaded into the PC
- CPU starts executing the **ISR**

Interrupt Handling

The **Ultimate** Interrupt Handler — ISR version of the **Ultimate Machine**

```
;-------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------

            ; Need to configure P1.2 for interrupts here

Inf_loop:   jmp     Inf_loop

;-------------------------------------------------------------------
; Interrupt Service Routines
;-------------------------------------------------------------------

P1_ISR:
            clr.b   &P1IFG          ;clear all interrupt flags
            reti                    ;return from interrupt


;-------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------
            .sect   ".reset"        ;MSP430 RESET Vector
            .short  RESET

            .sect   ".int37"
            .short  P1_ISR
```

Label of ISR

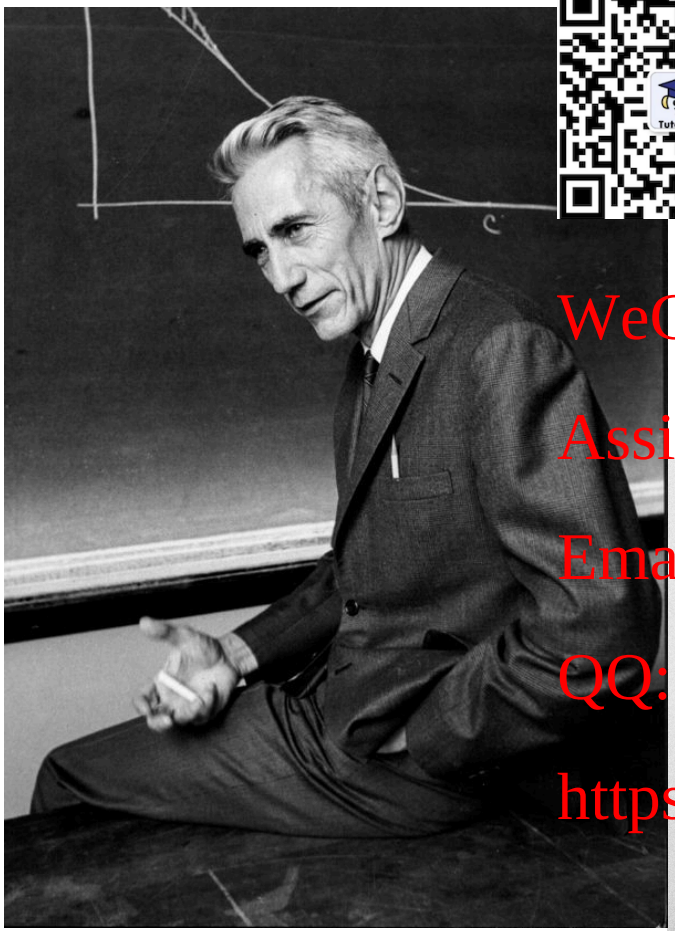
Label of ISR

ISR – similar to a subroutine
`reti` instead of **`ret`**
no **`call`** from main

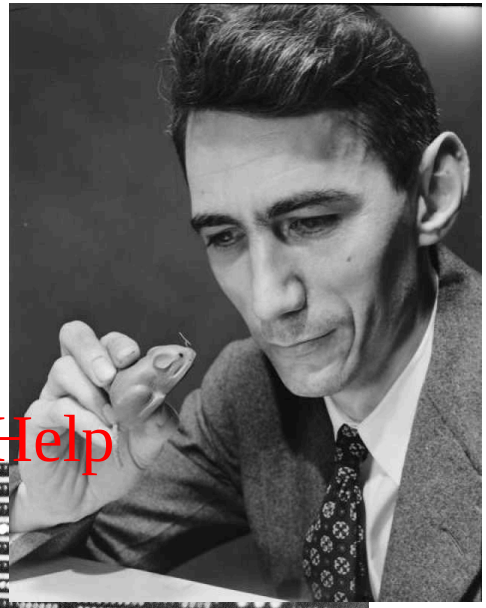
the ISR will be called from the **IVT**

The Ultimate Machine

A machine that does nothing – absolutely nothing – except switch itself off



Do you know this gentleman?



Interrupt Handling

CPU starts executing the ISR

- Unlike a subroutine, an ISR does not have input or output
- It can change global variables, can use the stack
- If the ISR is using the stack, it has to clean up the stack before `reti`
- Many interrupts are **multi-sourced** – e.g., both S1 and S2 trigger a flag in **P1IFG** and are served by the same ISR
- The ISR has to figure out which pin is implicated in the interrupt and perform the corresponding task
- The ISR must clear the interrupt flag it has served – otherwise, interrupt flags are not cleared and there will be a continuous interrupt cycle

Return from interrupt: `reti`

- Restores the Status Register from stack
- Restores the Program Counter from stack

Populating the IVT

slas789d.pdf

Table 6-4. Interrupt Sources, Flags, and Vectors (continued)

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
eUSCI_B1 receive or transmit)	UC... CTXIFG (SPI mode) UCB... IACKIFG, UCSTTIFG, UCS... CTXIFG0, UCRXIFG1, UCT... CTXIFG2, UCRXIFG3, UCT... CBIT9IFG (I²C mode) (UCB1IV)(1)	Maskable	0FFE2h	
DMA	DMA0CTL.DMAIFG, DMA1CTL.DMAIFG, DMA2CTL.DMAIFG (DMAIV)(1)	Maskable	0FFE0h	
Timer_A TA1	TA1CCR0.CCIFG	Maskable	0FFDEh	
Timer_A TA1	TA1CCR1.CCIFG to TA1CCR2.CCIFG, TA1CTL.TAIFG (TA1IV)(1)	Maskable	0FFDCh	
I/O Port P1	P1IFG.0 to P1IFG.7 (P1IV)(1)	Maskable	0FFDAh	
Timer_A TA2	TA2CCR0.CCIFG	Maskable	0FFD8h	

For a port P1 interrupt, the **word address for interrupt vector** is **0xFFDA**

When P1 raises an interrupt flag, execution will switch to the **label** that is given in this address

We need to write the label of the **ISR** that serves port P1 to this address

How?

Populating the IVT

How? We will use assembler directives

We start by finding the **wo** **ss for interrupt vector: 0xFFDA**

We locate the word addre **nker file "lnk_msp430fr6989.cmd"**

lnk_msp430fr6989.cmd | main.asm | main.asm | *main.asm | m

```
103     INT34                   : origin = 0xFFD4, length = 0x0002
104     INT35                   : origin = 0xFFD6, length = 0x0002
105     INT36                   : origin = 0xFFD8, length = 0x0002
106     INT37                   : origin = 0xFFDA, length = 0x0002
107     INT38                   : origin = 0xFFDC, length = 0x0002
108     INT39                   : origin = 0xFFDE, length = 0x0002
```


We add the **label of the ISR** to the Interrupt Vectors (at the end of *.asm)

```
;-----------------------------------------
; Interrupt Vectors
;-----------------------------------------
            .sect   ".int37"
            .short  P1_ISR

            .sect   ".reset"
            .short  RESET
```

Identifies address 0xFFDA

Label of ISR

Interrupts in Action: Blinky v. 2

Task: Make push button S1 blink the red LED

Every time push button S1 is pressed, the red LED toggles
i.e., if it is off, it turns on; if it is on, it turns off

Hint: Use interrupts

Interrupts in Action: Blinky v. 2

Configuration of the red LED and S1 in main

```
;-------------------------------------------------------------------------------
; Main loop here
;-------------------------------------------------------------------------------

            ; Need to configure red LED for output
            ; Red LED is at P1.0
            bic.b   #BIT0, &P1OUT       ; Red LED off
            bis.b   #BIT0, &P1DIR       ; Direction set to output

            ; Need to configure S1 for input
            ; S1 is connected to P1.1
            bis.b   #BIT1, &P1REN       ; Resistor enabled
            bis.b   #BIT1, &P1OUT       ; Pull-up resistor
            bis.b   #BIT1, &P1IES       ; Falling-edge triggers interrupt
            bis.b   #BIT1, &P1IE        ; Interrupt enabled

            ; Disable power lock
            bic.w   #LOCKLPM5, &PM5CTL0

            ; Enable general interrupts
            ; Put a nop before and after writing to SR to avoid warnings
            nop
            eint
            nop

main:       jmp     main
```

Interrupts in Action: Blinky v. 2

Interrupt service routine

```
;-------------------------------------------------------------------------
; Interrupt Service R
;-------------------------------------------------------------------------
P1_ISR:
            ; Check t                    the interrupt
            ; Is it p
            bit.b   #BIT1, &P1IFG
            jnc     return_from_P1_ISR

            ; P1.1 indeed, toggle red LED
            xor.b   #BIT0, &P1OUT       ; Toggle output 0 <=> 1

            ; Do not forget to clear the interrupt flag
            bic.b   #BIT1, &P1IFG

return_from_P1_ISR:

            reti                        ; return from interrupt

;-----------------------------------------
; Interrupt Vectors
;-----------------------------------------
            .sect   ".int37"
            .short  P1_ISR

            .sect   ".reset"
            .short  RESET
```