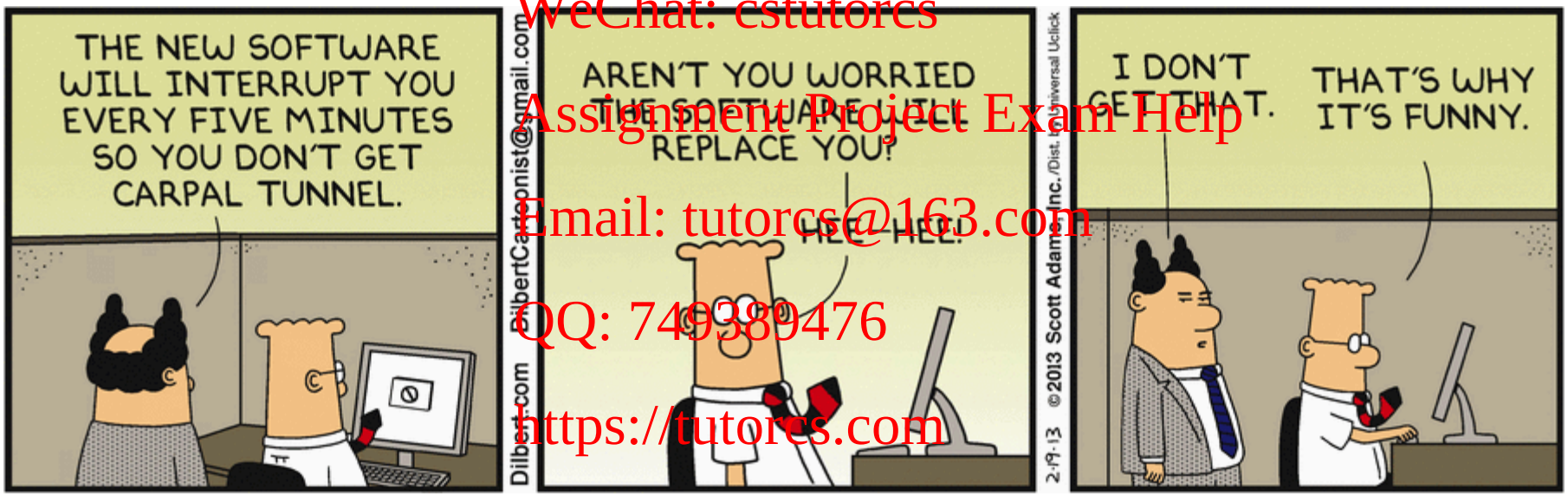
Lecture 22

Interrupts II



Joke of the Day

Why do programmers dark mode?

Because light attracts bugs.

Interrupts

What is an embedded system without interrupts?

Interrupts are a very effec[illegible]fficient way to deal with event that are **asynchronous** with the C[illegible]

- User input: e.g., through buttons
- Input from sensors: e.g., via I2C
- Timers: e.g., start of the next symbol in a communication system

The alternative to using interrupts is periodic or constant polling

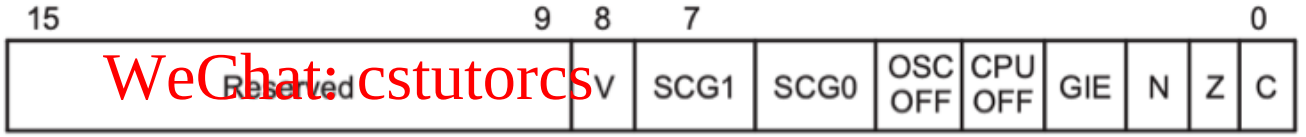
delay wasteful

In many embedded system applications, the MCU is idle (often in low power mode) waiting for interrupts

Recap: Interrupt Handling 1

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



- The highest priority interrupt is selected …
- ... its **Interrupt Service Routine (ISR)** is identified from the **Interrupt Vector Table (IVT)** … Programmer's responsibility to fill the **IVT**
- … address of the **ISR** is loaded into the PC
- CPU starts executing the **ISR** There is no explicit **`call`** to an ISR

Recap: Interrupt Handling 2

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

Legend:
Black: Runtime does it
Blue: Programmer does it

Configuring Px.y for Interrupts 1

Interrupt Edge Select Register: PxIES

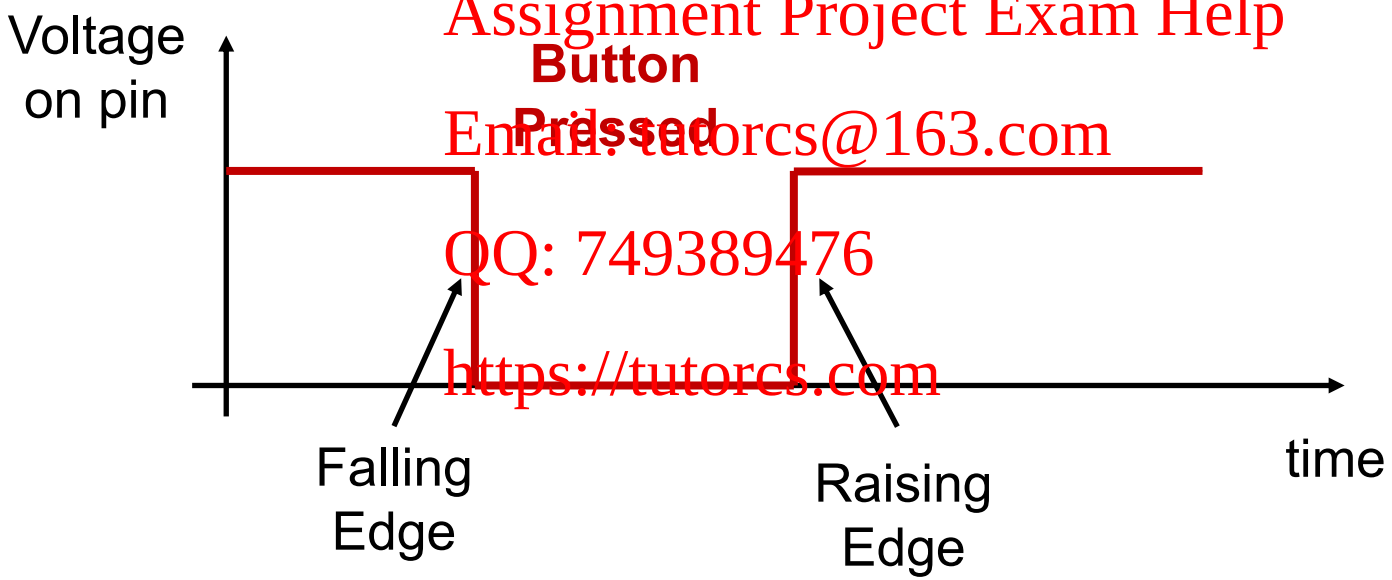
Bit **PxIES.y** selects the interrupt edge for pin **Px.y**

PxIES.y = 0: PxIFG.y flag is set on **a low-to-high transition**

PxIES.y = 1: PxIFG.y flag is set on **a high-to-low transition**

Raising Edge

Falling Edge



Configuring Px.y for Interrupts 1

Interrupt Edge Select Register: PxIES

Bit **PxIES.y** selects the in[illegible]ge for pin **Px.y**

Reading the manual slau[illegible]…

NOTE: Writing to PxIES

Writing to P1IES or P2IES for each corresponding I/O can result in setting the corresponding interrupt flags.

PxIES	PxIN	PxIFG
0 → 1	0	Will be set
0 → 1	1	Unchanged
1 → 0	0	Unchanged
1 → 0	1	Will be set

… you see that writing to PxIES can result in an inadvertent interrupt flag

Not great!

What to do? **Select interrupt edge before enabling interrupts!**

Also, clear interrupt flag after before next step

Configuring Px.y for Interrupts 2

Interrupt Enable Register: PxIE

Bit **PxIE.y** enables the **Px**[illegible]rrupt flag associated with pin **Px.y**

PxIE.y = 0: Inter[illegible] **Px.y** is **disabled** **by default**

PxIE.y = 1: Interrupt at pin Px.y is enabled

To enable interrupts for **Px.y** you need to **set the bit PxIE.y**

You also need to enable general interrupts in the Status Register SR

Several ways to do this

```
eint              ; enable general interrupts
bis.w  #GIE, SR   ; set GIE bit in SR
```


BIT3, defined in header file

Configuring Px.y for Interrupts 3

Interrupt Flag Register: PxIFG

Bit **PxIFG.y** is the interrupt flag associated with pin **Px.y**

PxIFG.y = 0: No interrupt is pending at pin **Px.y**

PxIFG.y = 1: An interrupt is pending at pin **Px.y**

If the interrupt is enabled

PxIFG.y is set by transitions, not static levels of input

Software, too, can set **PxIFG.y** to generate a software-initiated interrupt

The ISR is responsible for clearing **PxIFG.y** when it serves the interrupt

Again, mind your order of writing to port configuration registers

NOTE: PxIFG flags when changing PxOUT, PxDIR, or PxREN

Writing to PxOUT, PxDIR, or PxREN can result in setting the corresponding PxIFG flags.

Clear interrupt flag after before enabling interrupts

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)	UCB1CTXIFG (SPI mode) UCB1ACKIFG, UCSTTIFG, UCSTPIFG, UCTXIFG0, UCRXIFG1, UCTXIFG2, UCRXIFG3, UCBIT9IFG (I^2C 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