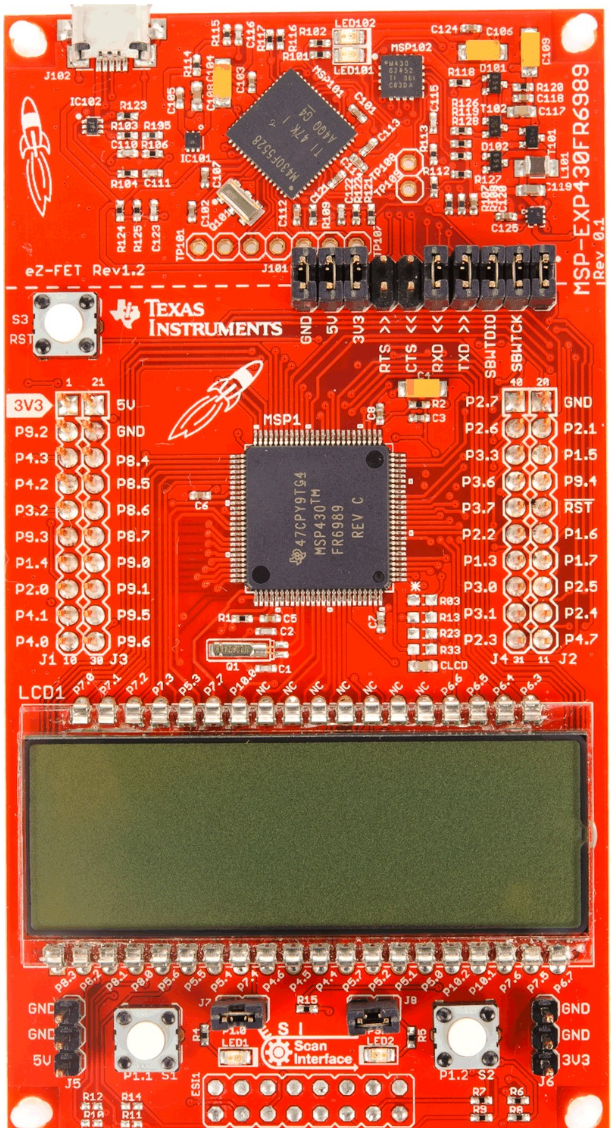


Red LED

Green LED

Lecture 20

GPIO II and Interrupts I

What's Next

Too many things to do, too little time

Good News:

No class on Friday November 10

No class Thanksgiving week – Wednesday 11/22 and Friday 11/24

Topics:

- Wrap up GPIO (Today)
- **Interrupts** (Starting today) GPIO Interrupts Blinky v.2
- Low power modes
- **Timers** and Timer Interrupts Blinky v.3
- Solutions Quiz5, Project etc.

Assignments:

- Quiz 6
- Midterm II – Teaching the MCU to count
- Final – Holiday Blinky

What's Next: Quiz 6

Will post **Quiz 6** after class today – **due Wednesday November 8**

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.

Why?

- Warm up for Midterm II – working with pushbuttons and LEDs, media recording etc.
- Good news: Might be the easiest assignment yet – single page
- And I will practically do half of it for you (at the end of Friday's class)

But First – Joke of the Day

Why do programmers always mix up Halloween and Christmas?

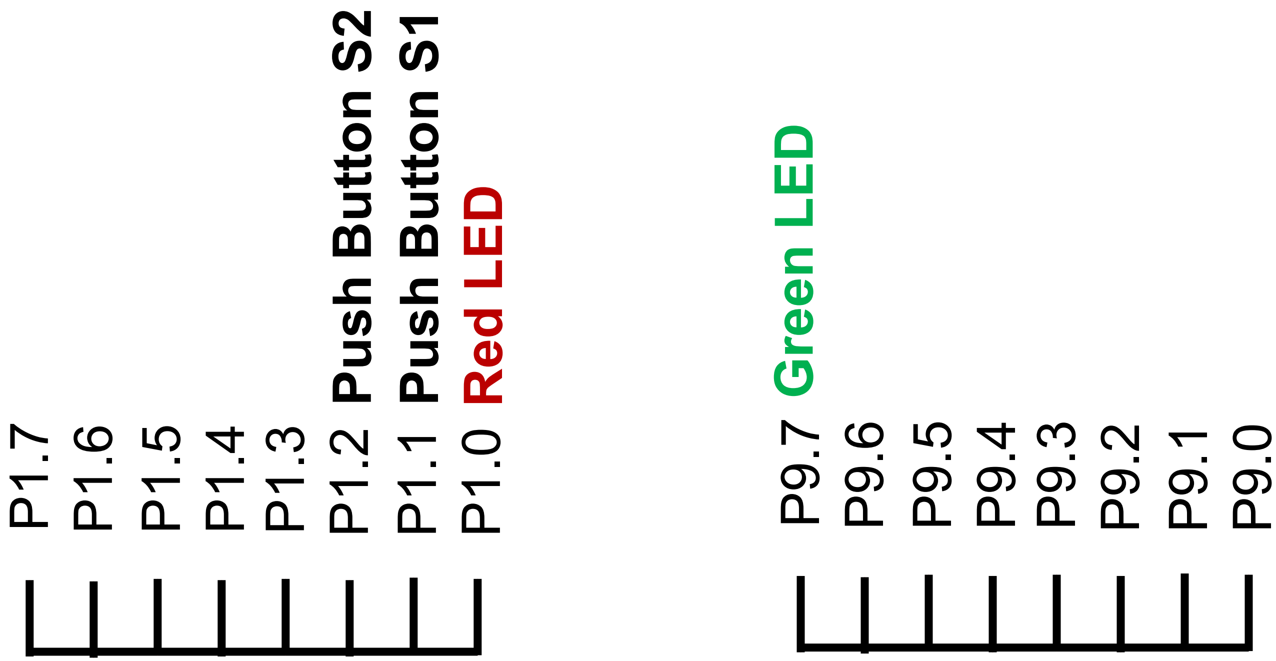
Because Oct 31 = Dec 25

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

Recap: GPIO Ports P1 – P10

Our MCU has 10 **General Purpose Input Output (GPIO) Ports P1 – P10**

- Each port has **8 pins** labeled as **Px.y** – **x** = port number, **y** = pin number
- The push buttons S1 and S2 and LEDs are connected to Ports P1 and P9



Pins of Port 1

Pins of Port 9

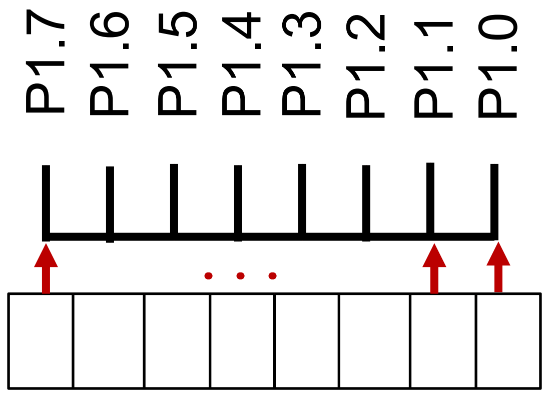
Recap: GPIO Ports Config Registers

Each port is configured and controlled by a set of **8-bit registers**

Px.y is controlled by **bit y** in the register corresponding to **port x**

e.g.

Pins



Bits

8-bit Registers

P1SEL0 / P1SEL1
P1SELC
P1DIR
P1OUT P1IN
P1REN
P1IFG
P1IE
P1IES

e.g., output HIGH on p1.0
$\Rightarrow$ set BIT0 in P1OUT
How?

```
bis.b   #BIT0, &P1OUT
```

How not to do it?

~~mov.b~~ #BIT0, &P1OUT

ALWAYS use bit operations
NEVER use move

Recap: Notation and Instructions

Shorthand notation:

PxDIR.y refers to bit **y** ∈ {0,1, …, 7} of register controlling port **x** ∈ {1, …, 10}

Px.y refers to pin **y** ∈ {0,1, …, 7} of port **x** ∈ {1, …, 10}

Instructions:

e.g.,

```
bic.b   #BIT0, &P1OUT
bis.b   #BIT0, &P1OUT
```

To check if a bit is 0 or 1

```
bit.b   #BIT0, &P1OUT
```

will set the carry bit if bit is 1

clear the carry bit if bit is 0

use `jc/jnc`

Recap: Configuring Px.y for Output

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

P1.0 **P9.7**

Step by Step Configuration for Output:

(**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**

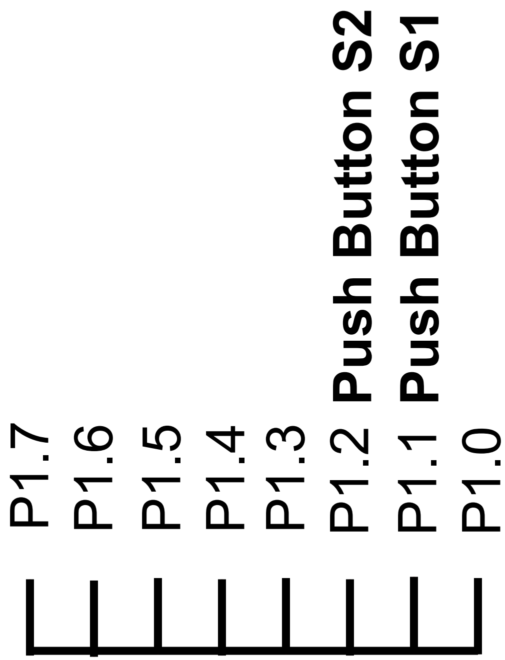
3. **Clear LOCKLPM5** – otherwise GPIO is not powered

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

Configuring Px.y for Input

Configuring a pin for input is more complex – requires **all** port configuration registers including **PxOUT with Role 2**

The only input we will use is push buttons S1 and S2



Pins of Port 1

Both are

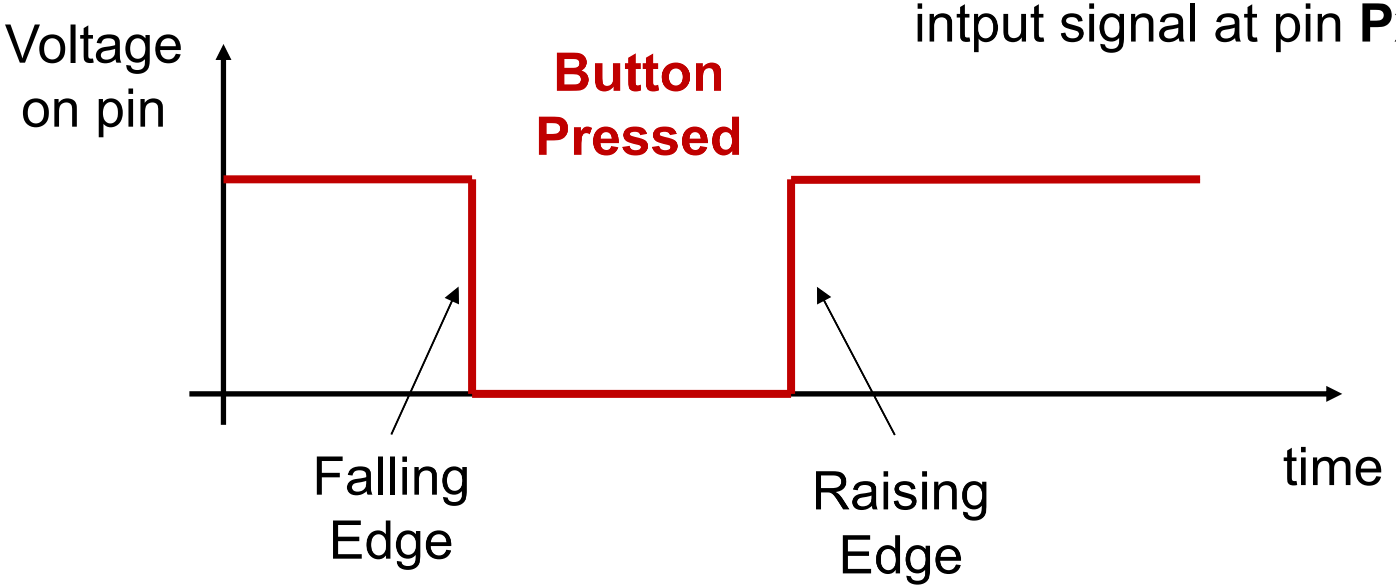
- Active low buttons
- Require a **resistor enabled**
- Resistor is in **pullup** configuration

Active Low Buttons

The push buttons S1 andS2 (and reset switch S3) are **active low buttons**

- when the switch is pressed/closed they send a LOW or "0" signal
- when the switch is open, they send a HIGH or "1" signal

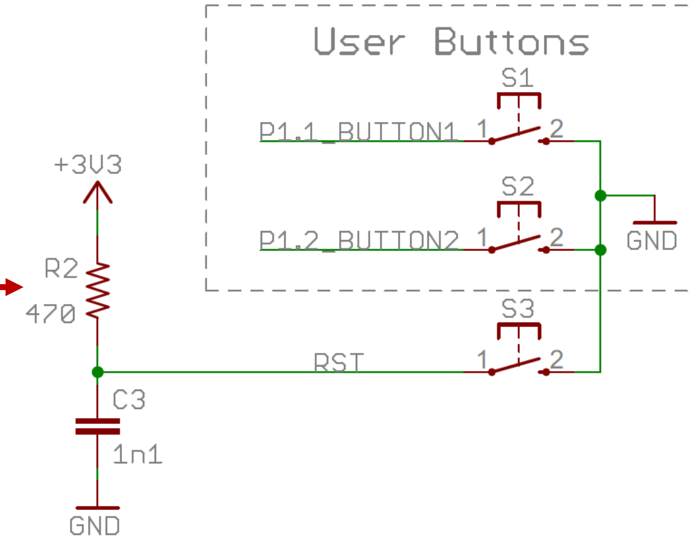
PxIN.y reflects the value of the intput signal at pin **Px.y**



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

Configuring the Resistor

Active low buttons require a **pullup resistor**



Pullup or Pulldown Resistor

Enable Register: PxREN

PxREN.y = 0: Resistor disabled **(default)**

PxREN.y = 1: Resistor enabled

need this

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

Output Register: PxOUT (Role 2)

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

PxOUT.y = 0: Pin **Px.y** is **pulled down (default)**

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

and this

Configuring P1.1 for Push Button Input

Step-by-step instructions

P1.1 is connected to push button S1

P1SEL0.1 = 0	Select GPIO functionality
P1SEL1.1 = 0	Default value is GPIO, no action required
P1DIR.1 = 0	Set pin direction to input Default value is input, no action required
P1REN.1 = 1	Enable resistor `bis.b #BIT1, &P1REN`
P1OUT.1 = 1	Configure for pullup resistor `bis.b #BIT1, &P1OUT`

Reading the Input at Pin Px.y

Input Register: PxIN

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

PxIN.y = 0: Input at pin **Px.y** is LOW

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

We will only deal with P1.1 or P1.2

Note: PxIN is a read-only register You cannot write to it.

How can we read the value?

```
bit.b  #BIT1, &P1IN
```

← sets the carry bit if bit is 1
clears the carry bit if bit is 0

When do we read the value?

No idea! The button can be pressed any time and we do not know when.

Reading the Input at Pin Px.y

The pressing of the push button is an **asynchronous event**

It can happen any time, is not tied to any clock of the MCU

There are two ways of dealing with it:

1. We constantly check the the `P1IN` register to see if the bit turns zero

   or at least periodically

   This approach is called **polling**

   very wasteful of resources

2. We let the push button trigger an **interrupt** ← **We will do this!**

There are three more port registers to configure/serve for interrupts

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

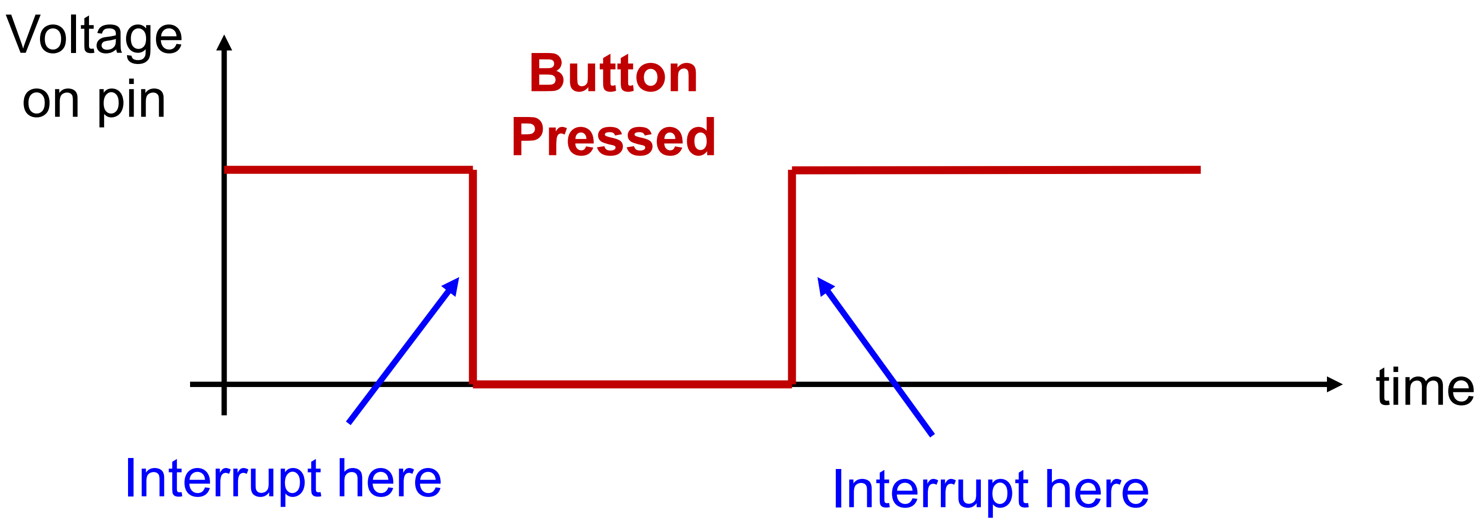
and GIE bit in SR

Push Buttons and Interrupts

We will not read the input signal (voltage on the pin) in **PxIN.y**

We will configure the push buttons to trigger interrupts

Either on a falling edge or raising edge



If **PxIES.y = 1**
Interrupt triggered
on falling-edge

If **PxIES.y = 0**
Interrupt triggered
on raising-edge

Configuring Px.y for Input/Interrupt

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

P1.1 **P1.2**

Step by Step Configuration for Input:

(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

Interrupts are events that temporarily suspend the normal execution of CPU

A peripheral (GPIO, eUSCI, a timer etc.) can raise a **flag** indicating that it needs immediate attention (pressed 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 stack as *empty*, PC points to the top of instructions

```
;
RESET       mov.w   #__STACK_END,SP         ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer


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


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

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

Flag

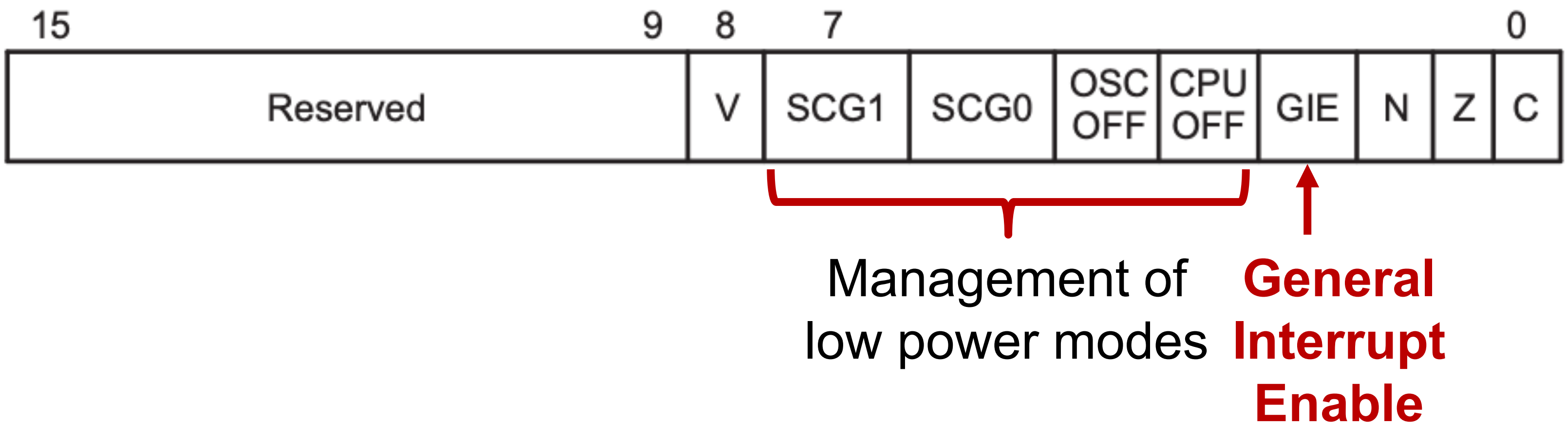
Corresponding ISR

Interrupt Vector Table

We will add more interrupt handlers to this list

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 execution of current instruction
- Program Counter **PC** is pushed onto the stack
- Status Register **SR** is pushed onto the stack
- SR is cleared

15 … 9	8	7							0
Reserved	V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C

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 S1 and/or S2 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

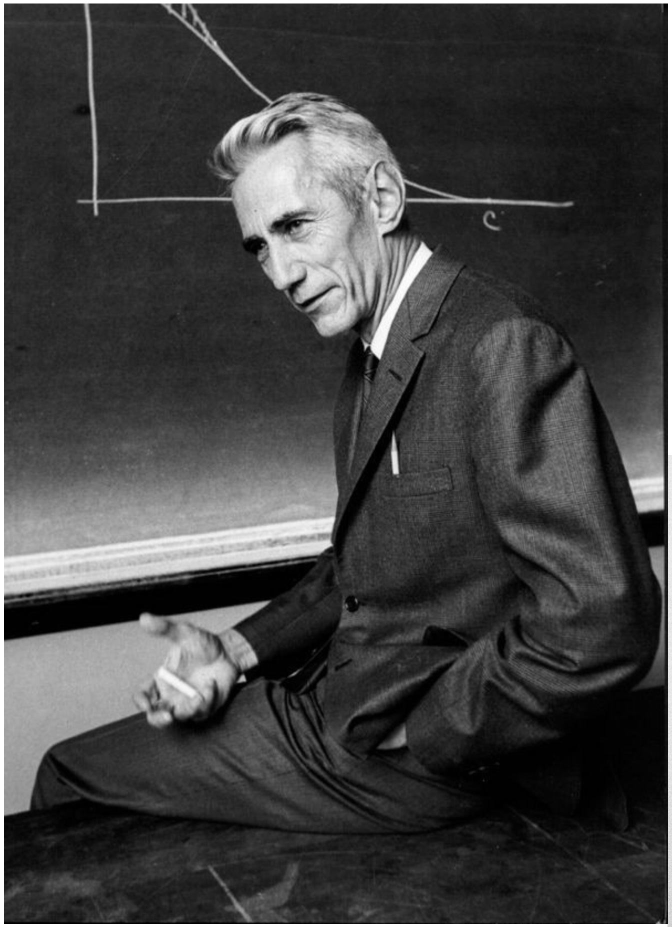
ISR – similar to a subroutine
`reti` instead of `ret`
no `call` from main

Label of ISR

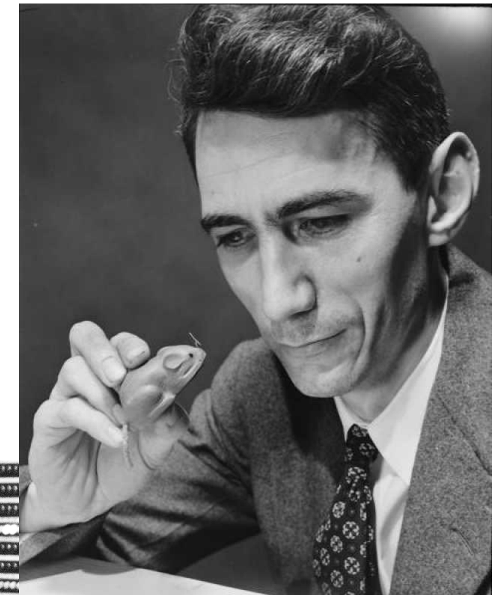
the ISR will be called from the **IVT**

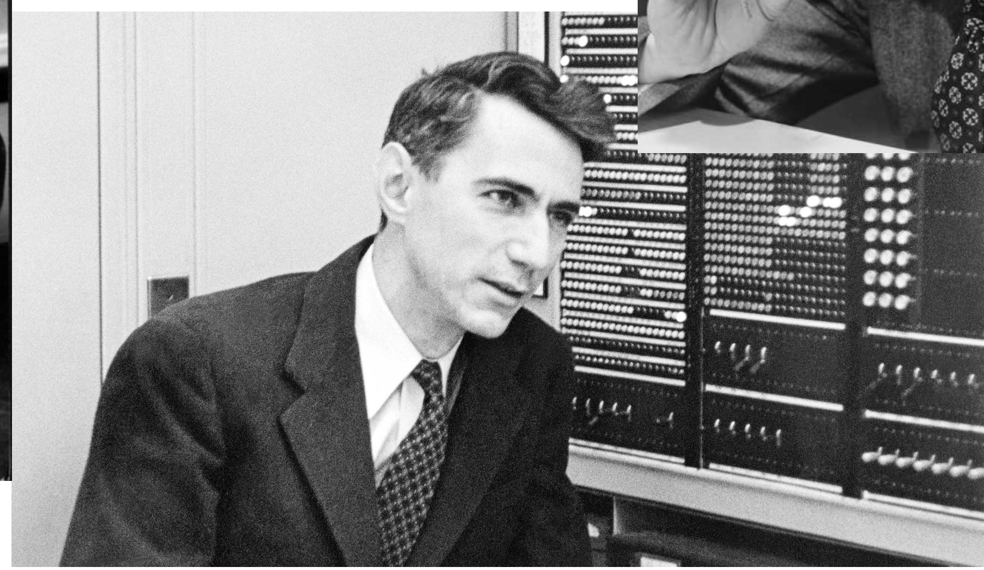
The Ultimate Machine

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, it 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