Lecture 20

GPIO General Purpose Input Output

But First – Joke of the Day

Why did NASA run Unix on the space shuttles?

Because you cannot open windows in space

What's Next: Project

Posted on Carmen – due Friday 3/31

You will write two subrout

```
;-------------------------------------------------------------------------
; Subroutine: inner_product_
;
; This subroutine takes two                rs of length n with Q-value m
;           and returns the innerproduct with same Q-value m
;
; Inputs: pointer to vector v1 in R7 -- can be modified
;         pointer to vector v2 in R8 -- can be modified
;         length n of v1 and v2 in R9 -- returned unchanged
;         Q-value 0<= m <15 in R10 -- returned unchanged
;
; Output: signed number in R13 -- R13 = v1.v2
;                                 where . denotes inner product
;
; All other core registers in R4-R15 unchanged
; No access to addressed memory
;-------------------------------------------------------------------------
```


Note that `R7` and `R8` contain the starting addresses of the vectors/arrays

How do you access the value that is at that address?

Recap: Indirect Register Modes

Indirect Register Mode of addressing works a charm here

Syntax

```
mov.w   @R7, R
```

Copy word from address

Indirect Autoincrement Register Mode works even better

Syntax

```
mov.w   @R7+, R5
```

Copy word from address in R7 to R5 then double increment R7

so it points to the next word in memory

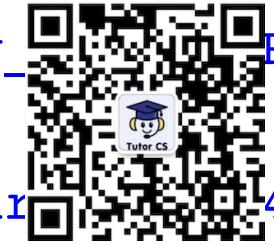
We have not indirect register modes so far because of two issues:

- Works for the source only, not destination
- Trickier to decide when to stop

Indexed vs Indirect Register Modes

Indexed Mode works for both source and destination

```
mov.w   array_[illegible]R5
...
mov.w   R5, ar[illegible]4)
```



Indirect Register Modes works for source only

```
mov.w   @R7, R5
...
mov.w   R5, 0(R7)
```


Question: How do we write to the memory location whose address is given in `R7`?

We use indexed mode: `0(R7)`

Indexed vs Indirect Register Modes

With **indexed mode** it is easier to determine when to end a loop

– just check the index

```
cmp.w   #LENGT
jlo     repeat
```

With **indirect register modes** there is no index, only addresses

Two options for determining loop termination

- Compute the address when you want to terminate or
- use a counter – e.g., if you want to repeat 64 times, initialize `Rx = 64`


```
mov.w   @R7+, R5
...
dec.w   Rx          ; account for one iteration
jne     repeat      ; repeat until counter hits zero
```

What's Next: Project

Second subroutine is signed multiplication

Will update the contracts confusion

```
;-------------------------------------------------------------------
; Subroutine: signed
;
; Inputs: signed word x in R5 -- returned unchanged
;         signed word y in R6 -- returned unchanged
; Note: abs(x) and abs(y) need to be <= 255 to avoid overflow
;
; Output: signed number in R12 -- R12 = R5 * R6
;
; All other core registers R4-R11 unchanged
; No access to addressed memory
;-------------------------------------------------------------------
```


Input to subroutine is signed **words** – but restricted in range

e.g., `R5 = 0xFFFF` i.e., x = -1

`R6 = 0xFFFE` i.e., x = -2

subroutine should return

`R12 = 0x0002`

What's Next: Project

Due date is Friday – **but office hours is Tuesday 1-3 pm**

You might not want to wa minute

Will post **Quiz 6** over the – **due Wednesday April 5 as promised**

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:

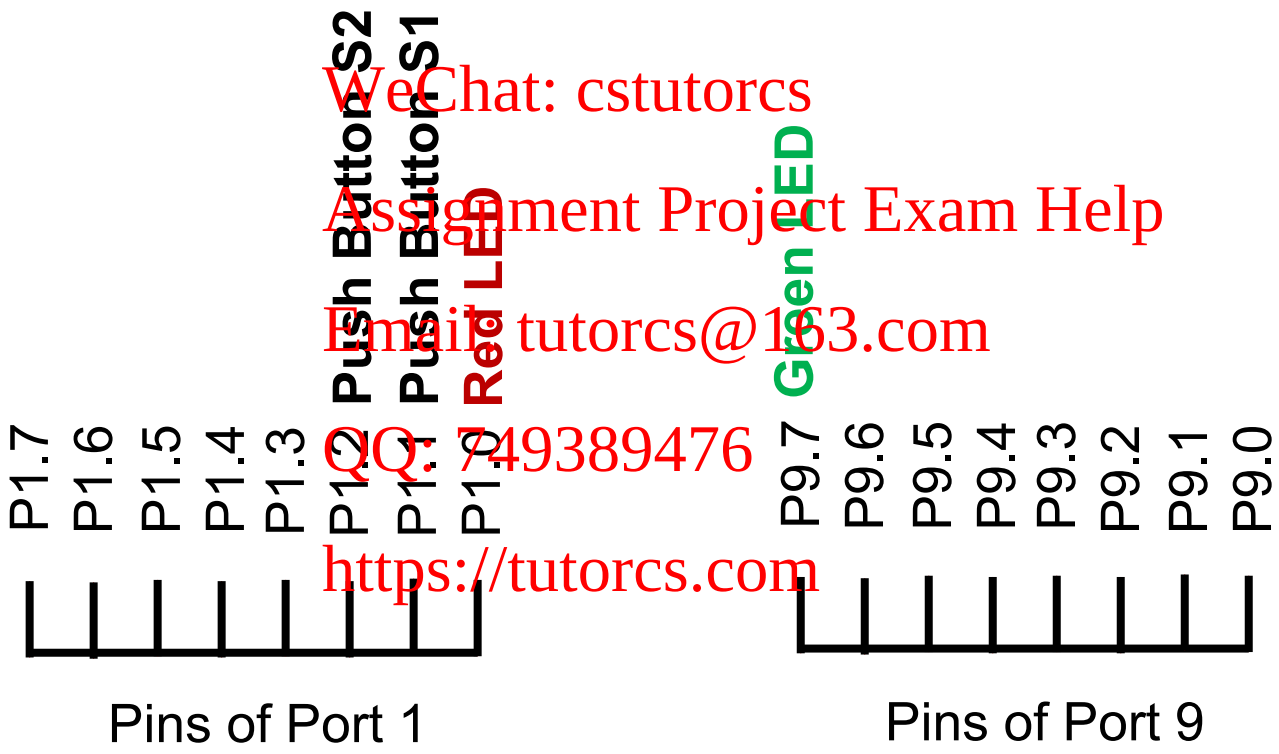
- Pressing S1 toggles the **green LED**
- Pressing S2 toggles the **red LED**

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.

Recap: GPIO Ports P1 – P10

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

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

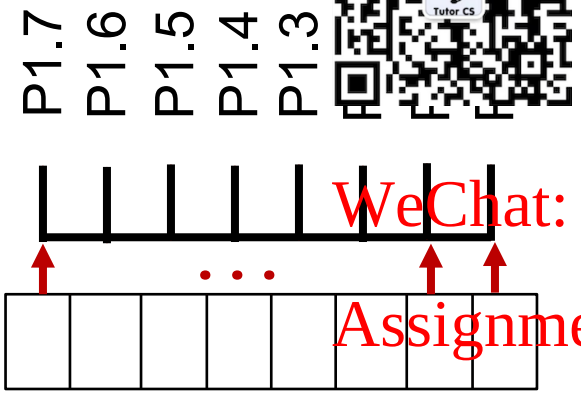


Recap: GPIO Ports Config Registers

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

Pin **Px.y** is controlled by **l** register corresponding to **port x**

e.g.



Use bit operations only – no `mov`

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

`byte` immediate mode

absolute mode

8-bit Registers

P1SEL0 / P1SEL1
P1SELC
P1DIR
P1OUT / P1IN
P1REN
P1IFG
P1IE
P1IES

```
#define P1IN
#define P1OUT
#define P1DIR
#define P1REN
#define P1SEL0
#define P1SEL1
#define P1SELC
#define P1IES
#define P1IE
#define P1IFG
```

Addresses are defined in the header file

Recap: Configuring Px.y

1. Select Pin Functionality: PxSEL0 and PxSEL1 (and PxSELC)

Default values are **PxSEL0.y = 0** and **PxSEL1.y = 0** for all x, y

⇒ The default function for Px.y is GPIO

⇒ No further action needed

2. Select Direction – Input or Output: PxDIR

PxDIR determines whether a pin functions as input or output pin

PxDIR.y = 0: Pin **Px.y** is switched to **input** direction **by default**

PxDIR.y = 1: Pin **Px.y** is switched to **output** direction

⇒ The default direction is input, you need to set the bit **PxDIR.y** when using **Px.y** for output, e.g. red and green LED

Recap: Configuring Px.y for Output

Configuring for output is simple:

1. **Set desired output v**
2. **Set direction to outp**

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

How do we set the output value?

Output Register: PxOUT.y is the value of the output signal at pin **Px.y** when the pin is configured as I/O function, **output** direction

PxOUT.y = 0: Output at pin **Px.y** is LOW

PxOUT.y = 1: Output at pin **Px.y** is HIGH

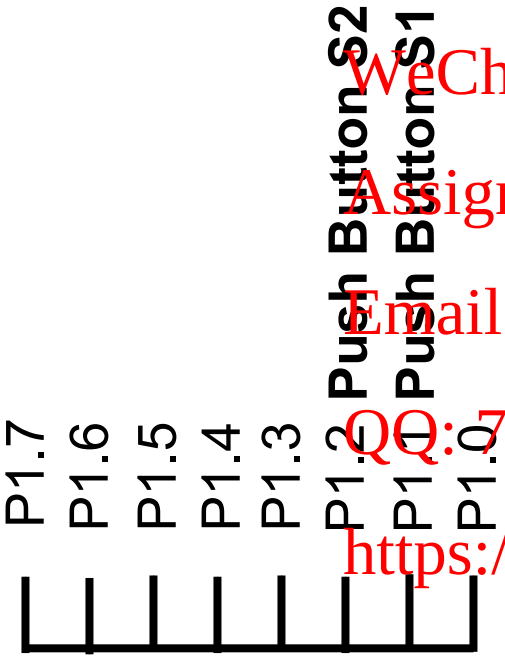
Example: Lighting up the red LED (Recall: red LED connected to P1.0)


```
bis.b   #BIT0, &P1OUT
bis.b   #BIT0, &P1DIR
```

Configuring Px.y for Input

Configuring a pin for input is more complex – requires **all** port configuration registers including **PxOU**[illegible]**e 2**

The only input we will use[illegible]uttons S1 and S2



Both are

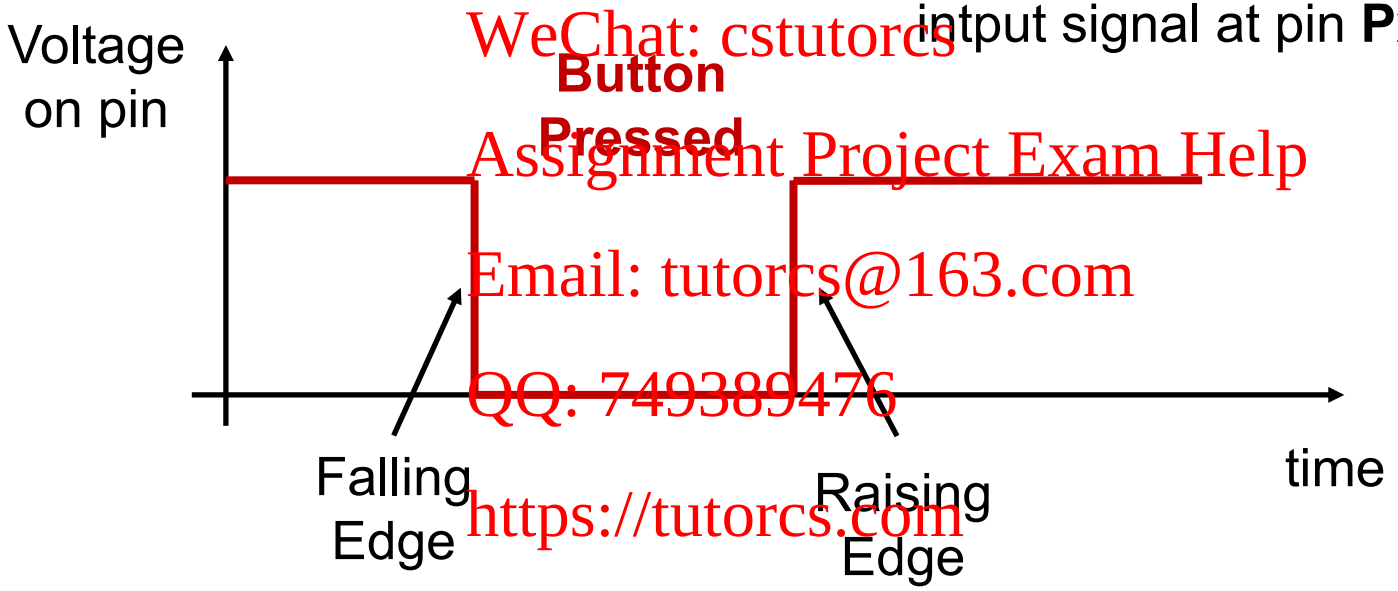
- Active low buttons
- Require a **resistor enabled**
- Resistor is in **pullup** configuration

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**



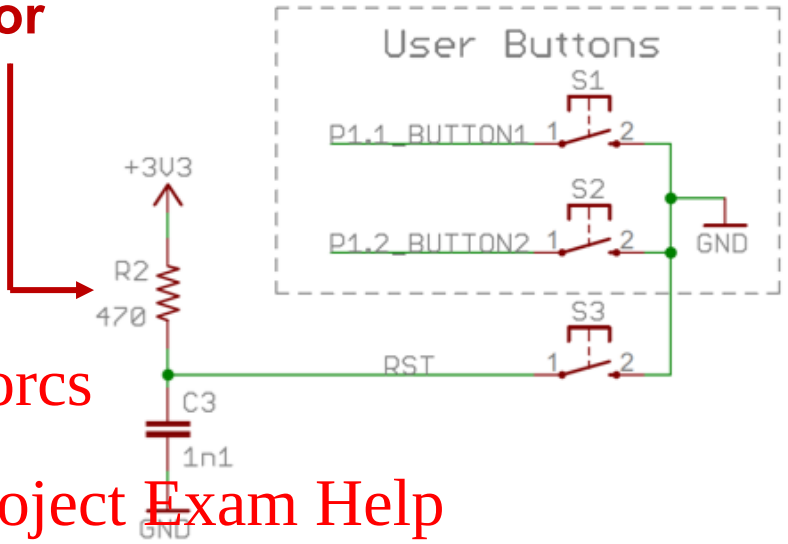
Spoiler: we will select falling or raising edge to trigger interrupts

Configuring the Resistor

Active low buttons require a **pullup resistor**

Pullup or Pulldown Res

Enable Register: PxREN



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

PxREN.y = 1: Resistor enabled

need this

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

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

Configuring P1.1 for Push Button Input

Step-by-step instructions

P1SEL0.1 = 0 S[illegible] O functionality

P1SEL1.1 = 0 D[illegible]ue 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 input 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

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

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

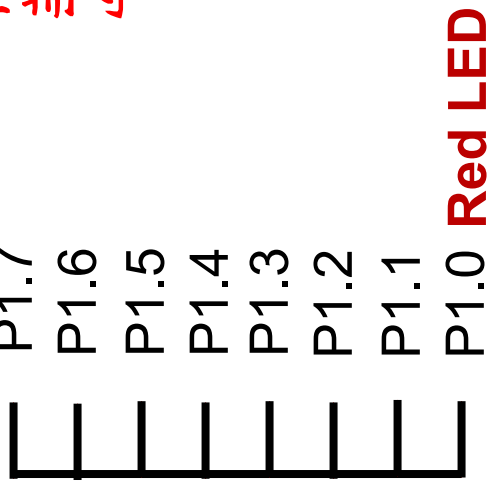
- PxIE – Interrupt Enable
- PxIES – Interrupt Edge Select
- PxIFG Interrupt Flag

GPIO in Action: Blinky v. 1

Task: Make the red LED blink

Go through documentation to find out:

- Red LED is connected to P1.0



- GPIO is default function for P1.y

⇒ No need to change P1SEL0 or P1SEL1

- For GPIO default is P1DIR.y = 0
- i.e., all pins P1.y are configures as input

⇒ Change P1DIR.0 = 1

- What about the output value?

⇒ Toggle it between HIGH and LOW P1OUT.0 = 1 and P1OUT.0 = 0

GPIO in Action: Blinky v. 1

Task: Make the red LED blink **Red LED is on P1.1**

How do we toggle betwee **P1OUT.0 = 1** and **P1OUT.0 = 0**?

```
xor.b #BIT0,
```


How about a timer?

⇒ Easiest way is to do a countdown timer

Start with a large unsigned value in a register

Decrease until the value hits zero

How do we get the LEDs to light up?

Need to enable GPIO output by clearing the LPM5 lock

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

GPIO in Action: Blinky v. 1


```
            bis.b   #BIT0, &P1OUT       ; First set output value
            bis.b   #BIT0, &P1DIR       ; Then change direction to output

            bic.b   #LOCKLPM5, &PM5CTL0 ; Override Power Lock

toggle:     xor.b   #BIT0, &P1OUT

            mov.w   0xFFFF, R5          ; Can omit this line – only first
                                        ; cycle will be of random length
countdown:  dec.w   R5
            jnz     countdown

            jmp     toggle
            nop

; The whole program is an extended infinite loop,
; no need to add anything here!
```


Exercise: Make the red and green LEDs blink in an alternating pattern.