ECE 2049 Lecture 8

Today

- More Digital I/O
- Intro to Lab 2

Office Hours

- Today: 2-6PM (Nick)
- Tomorrow: 3-5PM (Rusni)
- Monday: 3-5PM (Rusni)

Administrivia

- NW3: Due today by 11:59 PM
  - You can resubmit after class if you want
- NW4: Online after class, due Tues by 2PM
- Grades for NW1-3: this weekend
- Lab 1 Report: Due Fri by 11:59 PM
- Lab 2: Starts today
  - Prelab due by signoff or online submission by 6PM Tuesday

- EXAM 1: OUT AFTER TUESDAY'S CLASS
  (WHICH WILL BE REVIEW)
  - OPEN BOOK/NOTES/INTERNET/CALCULATOR (BUT INDIVIDUAL)
  - DESIGNED FOR 1-2 HRS
  - DUE THURS JUNE 16, BY 2PM (START OF CLASS)

- TOPICS: HW 1-4, LECTURES 1-8
  - NUMBER REPRESENTATIONS
  - MSP430 ARCHITECTURE
  - DIGITAL I/O
  - GENERAL CONCEPTS FROM LAB

2. **BONUS (2 pts)**: **Fun with memory-mapped peripherals:** Say you are you using a different microprocessor that exposes the memory bus so that you can add new memory-mapped peripherals to it. Assume that you attach your peripheral device and that it has one 16-bit value that the CPU can read at address 0x1104.

   How would you write code to read the value at this address? Like all register definitions in C, you can do this with a single `#define` statement. Complete the definition below, which includes an example of how the register should be used.

```
#define MY_REG    (/* Fill in your definition here! */)

void main(void)
{
      int val;
      val = MY_REG; // Read the value of the peripheral at address 0x1104

      // . . .
}
```

P2DIR = 5; // WRITE 5 TO THE ADDRESS WHERE P2DIR IS ON MEMORY BUS

// WANT TO WRITE TO MEMORY AT 0x1104

X = MY_REG

(* ((unsigned int *)(0x1104)))

DEREFERENCE

Digital I/O Examples

Example 1: Input and output registers

Assume the following digital I/O pins are configured correctly. P3.1-0 are configured as inputs, and P3.7-4 are outputs.

A Hypothetical Specification:

Input: Read a 2-bit binary value *a* on P3.1-0
Output: Given *a*, set P3.7-4 based on the table:

Input		Output			
P3.1	P3.0	P3.7	P3.6	P3.5	P3.4
a_1	a_0	z_3	z_2	z_1	z_0
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

"DECODER"

INPUT → LOGIC → OUTPUT

BUTTONS P3.1-0 — OUR CODE — LEDS P3.7-4

INPUT

READ P3.1-0 FROM INPUT REGISTER

CHAR INBITS = P3IN & (BIT1 | BIT0)

NEED TO SELECT ONLY BITS 1-0 FROM INPUT REGISTER

P3IN xxxx xxxx

& BIT1|BIT0 0000 0011 "MASK"

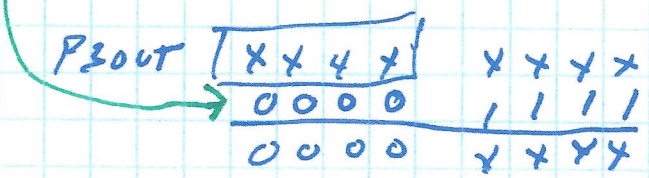
0000 00xx

ONLY BITS 1-0 CAN BE NONZERO

// SET ALL BITS IN OUTPUT (P3.7-4),
THEN TURN ON BITS WE NEED.

~~P3OUT = 0;~~

P3OUT &= ~(BIT7|BIT6|BIT5|BIT4);




```
SWITCH (INBITS)
{
  CASE 0:
    P3OUT |= BIT7; // SET P3.7 TO 1
    BREAK;

  CASE 1:
    P3OUT |= BIT6;
    BREAK;

  CASE 2:
    P3OUT |= BIT5;
    BREAK;

  CASE 3:
    P3OUT |= BIT4;
    BREAK;
}
```

NOW LET'S CONTINUE TO BUILD THIS EXAMPLE ON OUR LAB BOARD (WHICH USES DIFFERENT PINS)

Digital I/O Examples

Example 1: Input and output registers

Assume the following digital I/O pins are configured correctly. P3.1-0 are configured as inputs, and P3.7-4 are outputs.

SOME BUTTONS

A Hypothetical Specification:

Input: Read a 2-bit binary value *a* on P3.1-0

Output: Given *a*, set P3.7-4 based on the table:

4 LEDS

Input		Output			
P2.1	P1.1	P6.2	P6.1	P6.3	P6.0
a_1	a_0	z_3	z_2	z_1	z_0
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

INPUT → LOGIC → OUTPUT

INPUT: LAUNCHPAD BUTTONS.

OUTPUT: SOME LEDS

BUT NOW DO WE KNOW HOW/WHERE TO CONNECT TO PINS?

CONNECTING STUFF: KEY STEPS

1. FIND AVAILABLE PINS OR FIND WHERE HARDWARE IS CONNECTED (Px.y)
2. INPUT OR OUTPUT?
3. INTERPRET THE CIRCUIT (WHAT DOES A 1 OR 0 MEAN?)

(2.5. (IF INPUT) CONFIGURE PULL-UP/DOWN RESISTORS IF REQUIRED)

SEE DECODER EXAMPLE ON COURSE WEBSITE FOR A COMPLETE EXAMPLE W/ NOTES.

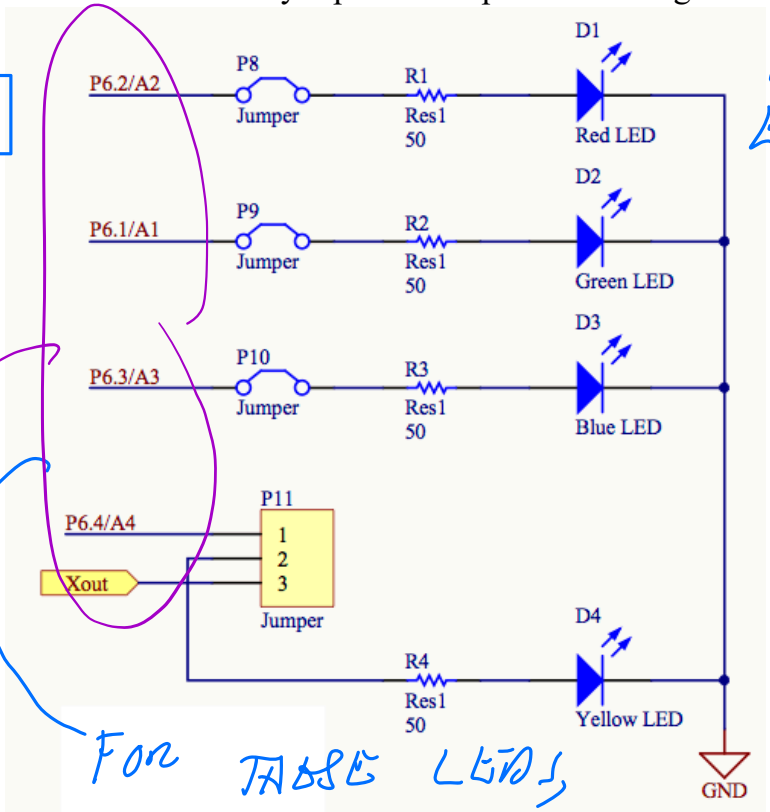
Digital I/O Concepts: Input or Output?

How do you know if something is an input or an output?

- If we are "reading" state from a hardware device, it is an **input**
- If we are "writing" or "setting" the state of a device, it is an **output**

BUTTONS ARE A TYPE OF INPUT ⇒ WANT TO GET THE STATE

Consider the LEDs on our board. Are they inputs or outputs? What logic level lights the LED?



6.2 | 6.1 | 6.3 | 6.4

LEDS ARE AN EXAMPLE OF OUTPUTS ⇒ WE SET THE STATE

FOR THESE LEDS, LOGIC 1 TURNS ON LED.

NOTE: THESE ARE NOT IN ORDER!

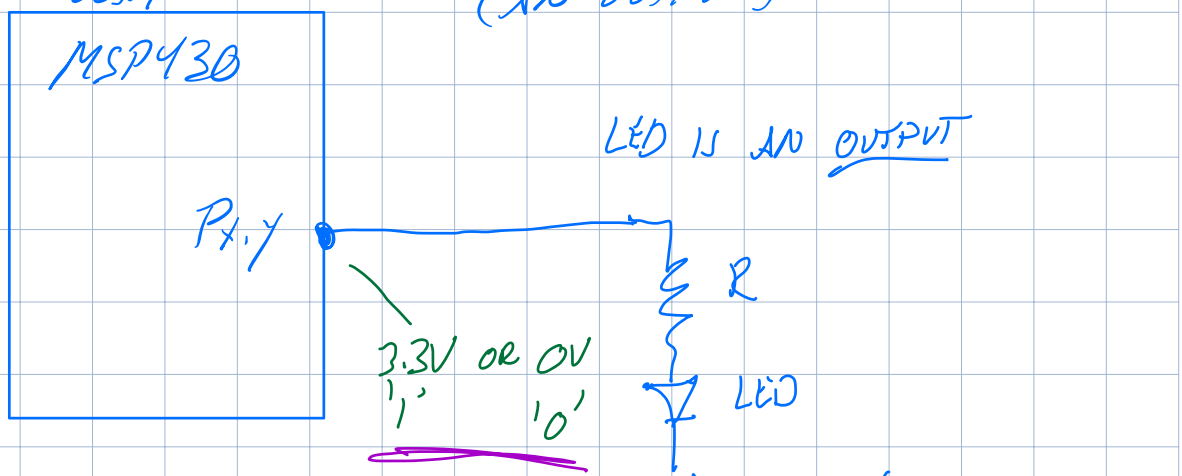
HARDWARE/SOFTWARE INTERFACE:
- WANT TO USE FUNCTIONALITY (SETLEDS() FUNCTION) WITHOUT CARING ABOUT WHICH PINS PERFORM WHICH FUNCTIONS!

In an application program like our demo project, we use the digital I/O ports repeatedly to use the buttons and LEDs. In these programs, it's a good idea to *wrap* the functionality for hardware components into useful functions.

See `setLeds()` in the demo project for an example!

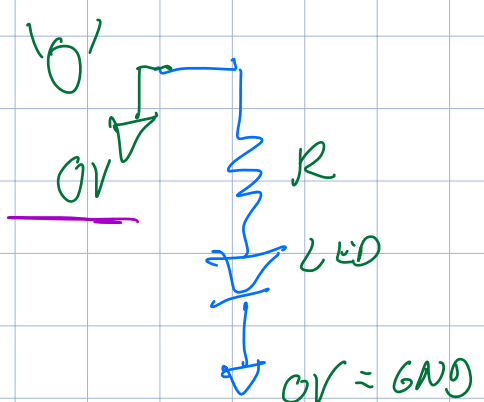
CONNECTING LEDs: A GENERIC PICTURE

ONE WAY: (AN OUTPUT)



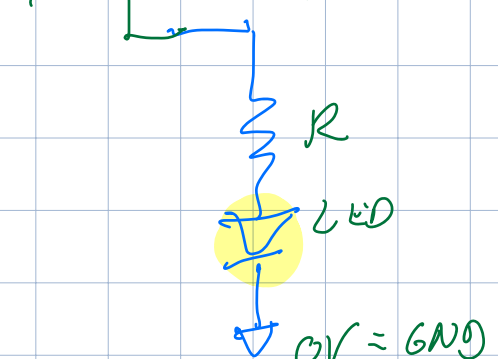
WHAT LOGIC LEVEL TURNS ON THE LED?

0 OR 1



OFF

(NO VOLTAGE DROP ACROSS LED)

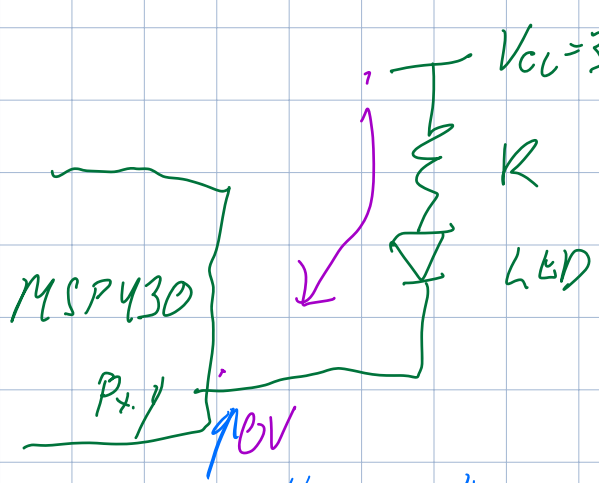


ON

(VOLTAGE DROP ACROSS LED)

COULD ALSO HAVE MANY OTHER CONFIGURATIONS!

Ex.

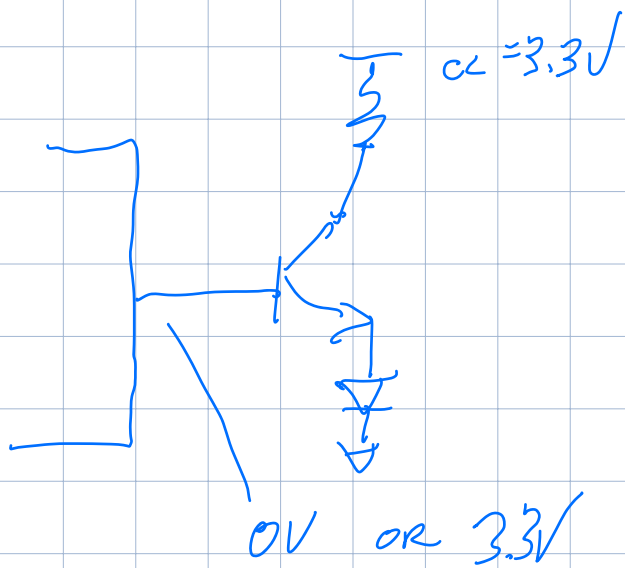


WHAT LOGIC LEVEL LIGHTS THE LED NOW??

'0'!

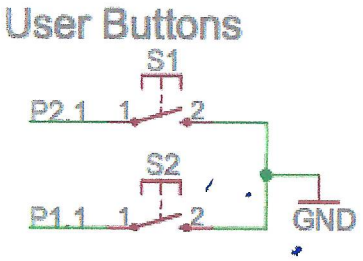
MSP430 "SINKS" CURRENT

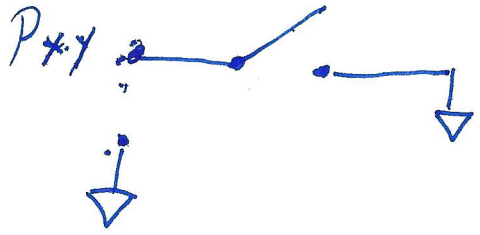
COULD ALSO USE A TRANSISTOR OR SOMETHING ELSE TO PROVIDE MORE POWER



Dealing with Inputs

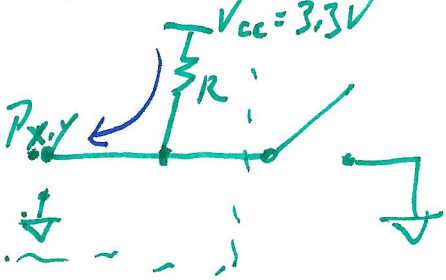
As we discussed briefly last lecture, inputs may require some special handling. Consider the buttons on the Launchpad board:





IN THIS CONFIGURATION IF SWITCH IS OPEN, INPUT IS UNDEFINED (FLOATING)!

FOR THIS CASE, USE A PULL-UP RESISTOR TO SET A DEFAULT STATE FOR THE PIN



WHEN SWITCH IS OPEN => 3.3V => '1'

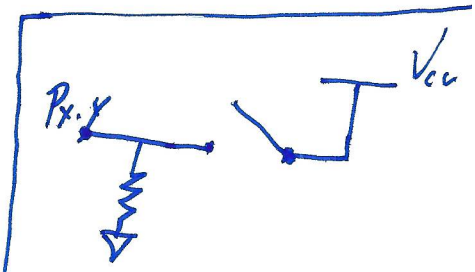
WHEN SWITCH IS CLOSED => 0V => '0'

PULL-UP RESISTORS ARE PROVIDED (OPTIONALLY) INSIDE THE MSP430 FOR INPUTS.

WHY?

DON'T NEED TO INCLUDE PULL-UP RESISTORS ON PCB

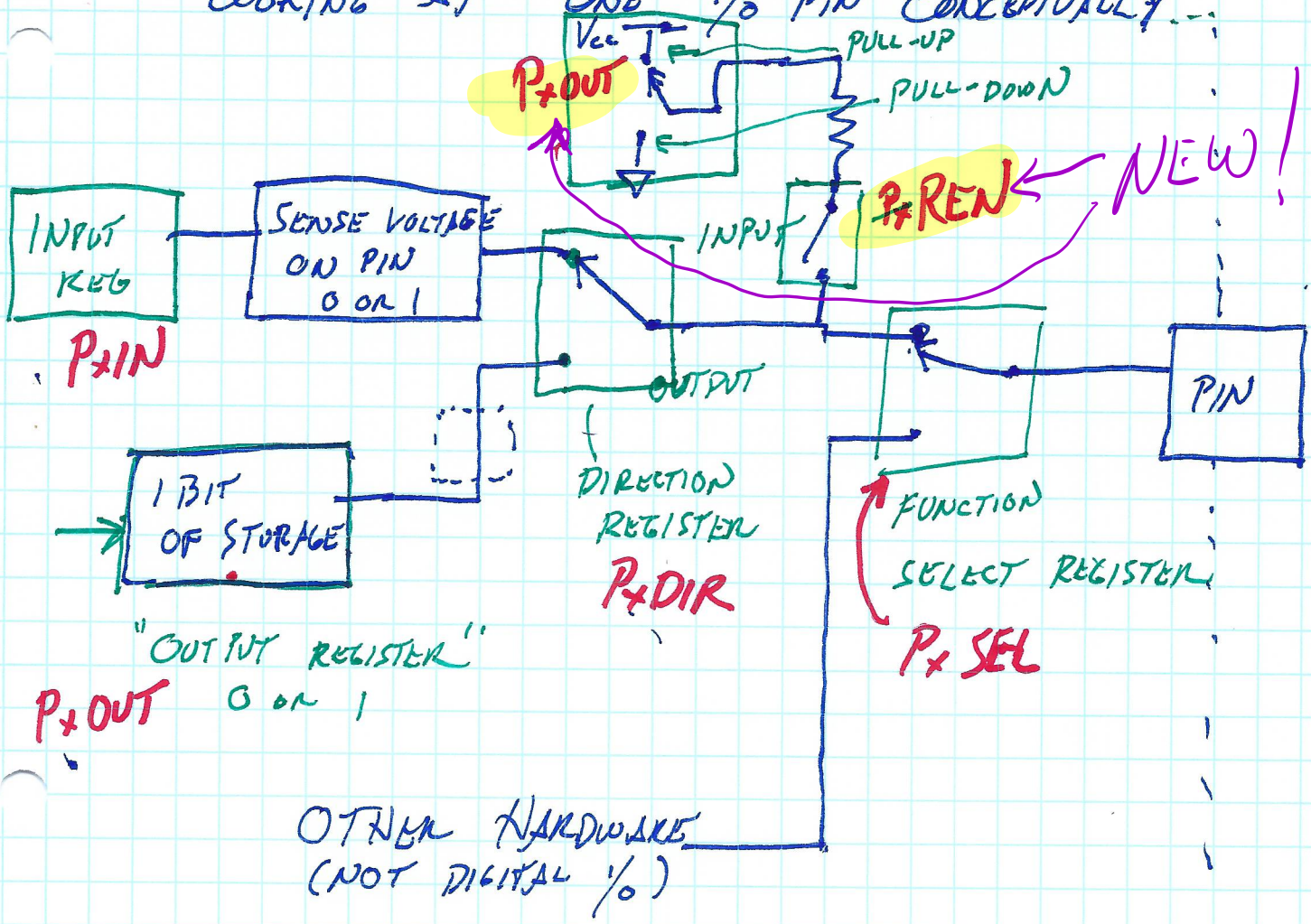
=> LOWER COST.



CAN ALSO HAVE A PULL DOWN RESISTOR, LIKE THIS.

SO WE HAVE SOME ADDITIONS TO THE HARDWARE FOR THIS... (Px REN)

LOOKING AT ONE I/O PIN CONCEPTUALLY...



Digital I/O Registers (cont.)

Pull-up/Pull-Down Resistor Enable (PxREN)

Activates pull-up or pull-down resistors when a pin is configured as a digital input.

1: ENABLE PULL-UP/DOWN RESISTOR

0: INTERNAL RESISTOR IS DISABLED

What controls whether to use a pull-up or pull-down resistor?

The output register (PxOUT) is actually re-used for this purpose!

Set the appropriate bits to 1 for pull-up resistors, and to 0 for pull-down. See p. 408 of the user's guide for details.

You will also see one more Digital I/O register…

Drive Strength (PxDS)

Controls "drive strength", or amount of current that is sourced from the pin when used as an output. We will always use the default setting for this.

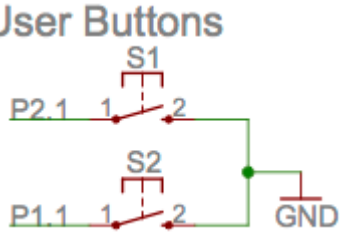
Set to 0 = Reduced drive strength (default) Set to 1 = Full drive strength

Important to note: all I/O pins have *limits* on the amount of current that can pass through them (usually on the order of milliamps). See the MSP430F5529 datasheet for details.

USUALLY ≈ 10mA

As an embedded developer, it's always important to remember the requirements of the hardware as well as the software!

Example: Launchpad Buttons (cont.)



We can configure these buttons as inputs and using pull-up resistors, as follows:

```
void initButtonsLecture(void)
{
      // Configure buttons as outputs using internal pull up resistors
      // Button 1:  P1.1
      P1SEL &= ~BIT1;
      P1DIR &= ~BIT1;
      P1REN |=  BIT1;
      P1OUT |=  BIT1;

      // Button 2:  P2.1
      P2SEL &= ~BIT1;
      P2DIR &= ~BIT1;
      P2REN |=  BIT1;
      P2OUT |=  BIT1;
}
```

← SELECT FOR DIGITAL I/O
← SET AS INPUT
← CONFIGURE FOR PULL-UP

Note that the buttons are on different ports, so we need to configure them separately!

```
// Read buttons S2 and S1 and return their state in the
// lower two bits of the return value such that
// ret = 0 0 0 0    0 0 S2 S1
unsigned char readButtonsLecture(void)
{
                SEE READLAUNCHPADBUTTONS()
                IN PERIPHERALS.C IN
                   LAB 2 TEMPLATE.


}
```

Polling

How can you monitor and use your properly-configured digital I/O functions?

- … by repeatedly checking if the button status has changed!
- Since this just involves reading a memory address, it is very fast to execute (on the order of microseconds!)

Example:

```
// Inside your demo project...
while(1)
{
    ret_val = readButtons();
    setLeds(~retVal);
}
```

Another, similar example:

```
ret_val = 0x0f; // Default value for all buttons unpressed
while(ret_val == 0x0f)
{
    ret_val = read_buttons();
}

setLeds(~ret_val);
```

Without a delay, this loop executes in microseconds!

This process is called **polling**–we constantly check the buttons and do something when they change.

At the moment, it's all the program needs to do, so it's fine. But what if we wanted to perform more tasks? What if we wanted the processor to sleep while it was waiting?