CE 2049 Lecture 5

Office Hrs

- Nick: 5-7PM Today
- Jonathan: 2-4PM Wed
- Nick: 5-7PM Thurs

Administrivia

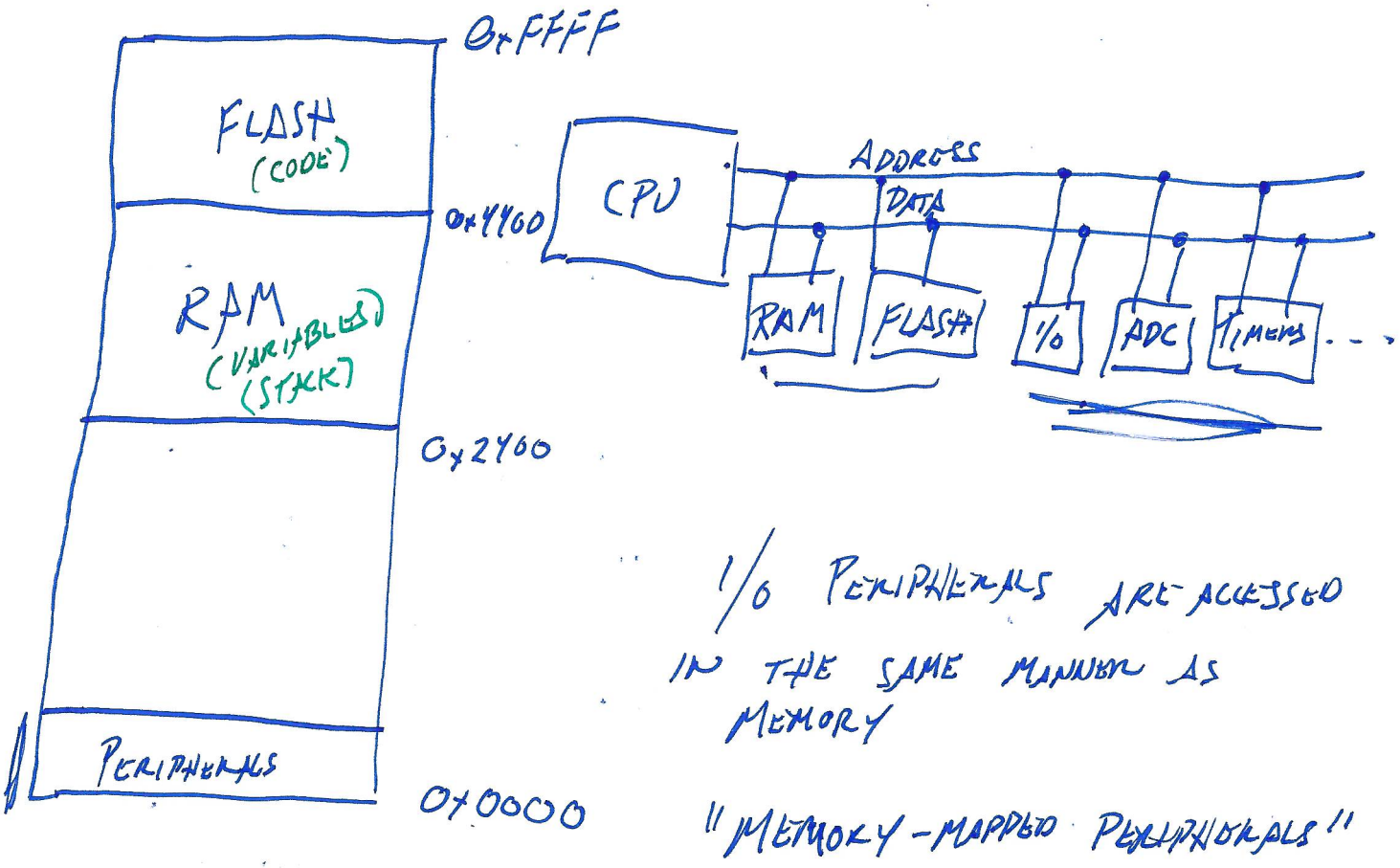
- HW 2: Due Today
- Lab 0: Report Due 11:59PM Tonight
  - Submit code + Report on Canvas
  - Can obtain signoff during office hrs (or contact me) if you still need it
- Lab 1 starts Thursday
  - Prelab due Thursday (6/10) by 11:59PM EDT
    - via "signoff" or online submission
  - Fill out survey to be matched w/ a "lab group" by Wed at 11:59PM EDT
    - Discuss ideas, share designs, debug things, but you are still responsible for the whole lab
    - Everyone submits their own code, report, + does separate signoffs
- HW 3: Online Today
- Exam 1: ~~Oct~~ Next Tues (details soon) (HW 1-3)

CONTINUING FROM LAST TIME...

Mapping Memory

In practice, the "memory space" is *mapped* across the different types of memory and hardware devices connected to the CPU.

- This includes the different types of physical memory (RAM, flash), as well as *hardware peripherals*
- The mappings of which components use which addresses is based on the physical wiring in the IC (we know the mappings based on the header files)



The Memory Map (found in MSP430F5529 datasheet)

Table 5. Memory Organization(1)

		MSP430F5522 MSP430F5521 MSP430F5513	MSP430F5525 MSP430F5524 MSP430F5515 MSP430F5514	MSP430F5527 MSP430F5526 MSP430F5517	MSP430F5529 MSP430F5528 MSP430F5519
Memory (flash) Main: interrupt vector	Total Size	32 KB 00FFFFh–00FF80h	64 KB 00FFFFh–00FF80h	96 KB 00FFFFh–00FF80h	128 KB 00FFFFh–00FF80h
Main: code memory	Bank D	N/A	N/A	N/A	32 KB 0243FFh–01C400h
	Bank C	N/A	N/A	32 KB 01C3FFh–014400h	32 KB 01C3FFh–014400h
	Bank B	15 KB 00FFFFh–00C400h	32 KB 0143FFh–00C400h	32 KB 0143FFh–00C400h	32 KB 0143FFh–00C400h
	Bank A	17 KB 00C3FFh–008000h	32 KB 00C3FFh–004400h	32 KB 00C3FFh–004400h	32 KB 00C3FFh–004400h
RAM	Sector 3	2 KB(2) 0043FFh–003C00h	N/A	N/A	2 KB 0043FFh–003C00h
	Sector 2	2 KB(3) 003BFFh–003400h	N/A	2 KB 003BFFh–003400h	2 KB 003BFFh–003400h
	Sector 1	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h
	Sector 0	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h
USB RAM(4)	Sector 7	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h
Information memory (flash)	Info A	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h
Bootstrap loader (BSL) memory (flash)	BSL 3	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h
Peripherals	Size	4 KB 000FFFh–0h	4 KB 000FFFh–0h	4 KB 000FFFh–0h	4 KB 000FFFh–0h

(1) N/A = Not available
(2) MSP430F5522 only
(3) MSP430F5522, MSP430F5521 only
(4) USB RAM can be used as general purpose RAM when not used for USB operation.

What can we learn from this?

- RAM starts at 0x2400, implemented in 2KB "Banks"
- Flash uses the address range 0x4400 to 0xFFFF
  - Code is written to flash starting from this address
- What about addresses 0x0010-0x0fff?

So what's the deal with addresses 0010h-0FFFh again?

These addresses are assigned to *peripherals:*

- Each peripheral has its own registers that are *mapped* as part of the memory that the CPU can access
- CPU can read or write data to peripherals just like any other memory address

This is how you make the CPU do I/O!

→ #inc MSP430.H

HARDWARE

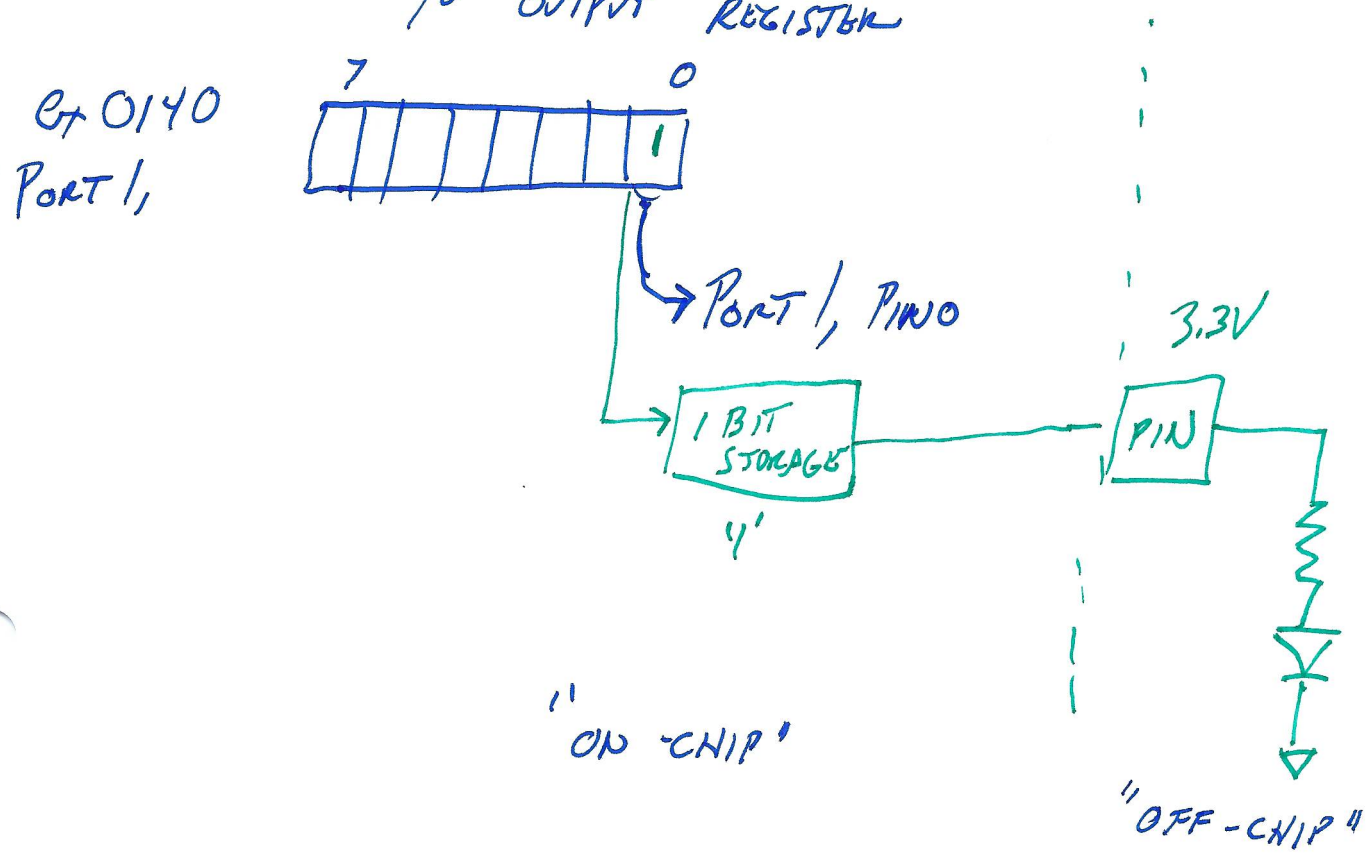
REGISTERS: - LOCATIONS IN MEMORY THAT ARE MAPPED TO PERIPHERALS

- 8 BIT
- 16 BIT

- EACH ONE HAS SPECIAL MEANING FOR THAT HARDWARE COMPONENT

EX.

DIGITAL I/O "OUTPUT REGISTER"



Input and Output

Consider this C code for a general-purpose system:

```
#include <stdio.h>

void main()
{
   char inKey = '-';  // declare variable named inKey
                      // and initialize to ASCII '-'

   while (inKey != 'X');
   {
         /* get character from keyboard */
         inKey = getchar();

         /* display character entered on screen */
         putchar(inKey);
   }
}
```

What is really happening here?

`getchar()` and `putchar()` are functions from the C standard library (part of stdio.h)

- Library for these functions is part of OS, and linked into code during build process
- These functions have always been part of the standard library because general purpose systems have always needed to use this type of I/O (eg. keyboard, screen, …)

Example: When a key is pressed, several layers below our little application, a byte has been placed on the microprocessor's data bus from a *port* connected to the keyboard:

Module 5. Digital I/O

Topics

- More Digital I/O

About Digital I/O

Why do we use Digital I/O anyway?

Digital I/O is a method of directly inputting our outputting logic levels to the pins of the MSP430 Package.

You can use this functionality to implement almost anything!

- Simple devices: Buttons and LEDs
- Control signals for complex peripherals
- … and more!

0 → "LOGIC 0" → 0V
"LOGIC LOW"

1 → "LOGIC 1" → 3.3V (FOR US)
"LOGIC HIGH"

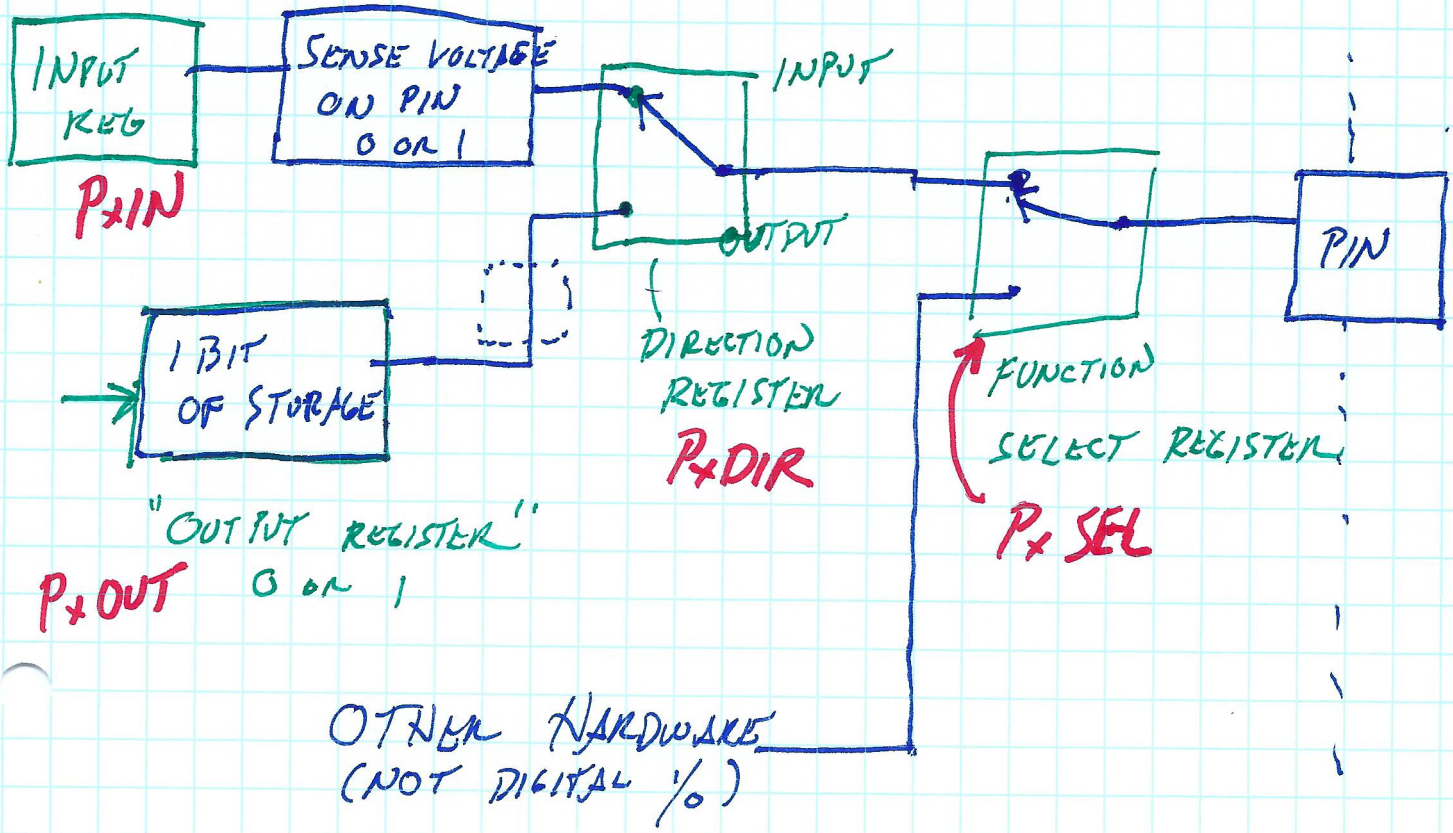
→ CONNECT PHYSICAL PINS TO MEMORY

– "READ" 0 OR 1 FROM A PIN => INPUT

– "WRITE" 0 OR 1 TO A PIN => OUTPUT

~~CASE~~

LOOKING AT ONE I/O PIN CONCEPTUALLY...



Fun Facts about Digital I/O

- Eight independent, individually-configurable *ports*, named P1-P8
- Ports 1-7 each have 8 configurable *pins*, and are thus 8 bits wide; Port 8 is 3 bits wide
  Pins are referenced as P<port>.<pin>, eg. P1.4.
  ↳ P1.4 = PORT 1, PIN 4
- Each pin of each port can be configured individually as input or output
- Most digital I/O pins share physical *package pins* with some other function on the device. This is called *pin multiplexing*.
- Each port is controlled by **six** single-byte **registers**

What is a register, anyway?

Register: CIRCUIT SIMILAR TO MEMORY, BUT W/ SPECIAL FUNCTIONALITY IN HARDWARE

- Registers have addresses just like standard memory, so you can read and write to them
- Provide interface between hardware and software:
  - Reading from a register can get information about the hardware
  - Writing to a register can change how the hardware is configured, or send information to a component
- Functionality provided is defined by the hardware's design. When TI designs the MSP430, they define what registers are exposed to the programmer, which defines the functionality available on the chip.
- All the I/O port registers are *memory-mapped*: each register associated with a digital I/O port has a unique address in memory
  - How do you know what the addresses are? These are defined in the MSP430F5529 datasheet, as well as msp430.h and msp430f5529.cmd

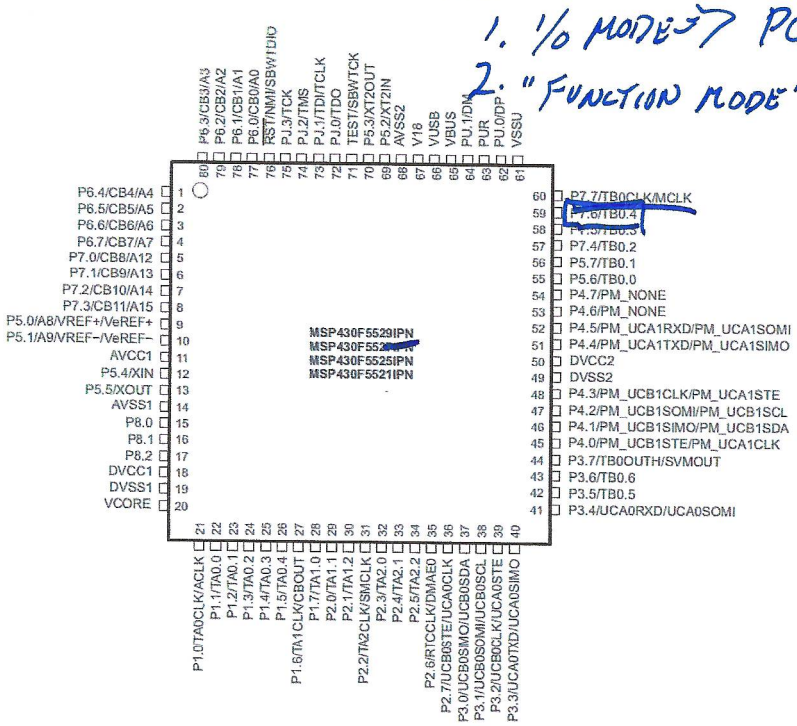
Pins on the Microcontroller

Microcontrollers often pack lots of functionality in a small IC. However, the usage of all this functionality is limited by the physical pins on the IC *package*:

PIN 59: P7.6/TB0.4

TWO FUNCTIONS

1. I/O MODE => PORT 7, PIN 6
2. "FUNCTION MODE" => TIMER B0 INPUT 4



In order to maximize the usage of physical pins, most physical pins (also called "package pins") are shared between multiple device functions.

=> PIN MULTIPLEXING

- ONE PART CAN OFFER MORE FUNCTIONALITY THAN IT HAS PINS
- UP TO DESIGNER TO PICK FUNCTIONALITY IN SOFTWARE.

Digital I/O Registers

The 6 registers controlling the digital I/O ports are as follows. **Each bit of the register controls the state for a specific pin.**

EX. PORT 3

Function Select Register (PxSEL) P3SEL

Selects the port pin for Digital I/O–remember multiplexing? This selects the function used on the pin.

CONTROL REG

SET TO 0: SELECT DIGITAL I/O MODE

SET TO 1: SELECT "FUNCTION MODE"

Direction Register (PxDIR)

Sets port pins as Input or Output

Set to 1 = Output

Set to 0 = Input

EX. P5DIR = 0x0F; 7654 3210
0000 1111

P5.7-4 ARE INPUTS

P5.3-0 ARE OUTPUTS

Input Register (PxIN)

This is where the value input on the port appears (this is where you "read" the port)

IF PIN IS AN INPUT

CHAR v = P3IN; READ INPUT REG, STORE INTO v

Output Register (PxOUT)

This is where data to be output on the port should be "written"

IF PIN IS AN OUTPUT

P3OUT = 0xAA; 7 6 5
1010 1010

Drive Strength (PxDS)

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

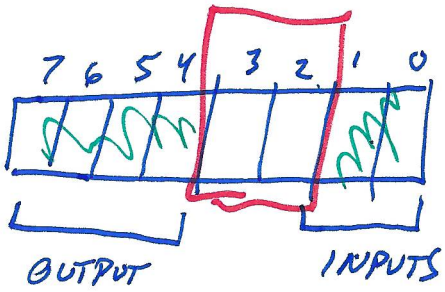
We will discuss these two (using examples) later.

Conceptually, once you know which registers to use, using Digital I/O is pretty simple–all you need to do is read or write the desired values to the registers.

Configuration Example

Example: Configure Port 3 for Digital I/O with pins 1 and 0 as inputs and pins 7-4 as outputs.

There are two ways we can approach this problem:



STEPS

1. SET PINS FOR DIGITAL I/O MODE. => P3SEL
2. SET PINS AS INPUT OR OUTPUT => P3DIR

ONE WAY (BAD WAY):

```
P3SEL = 0;      // ALL PORT IS IN I/O MODE
P3DIR = 0xF0;   // 1111 0000
                   ↑OUTPUT ↑INPUT
```

IN THIS CASE, PINS 3-2 ARE OVERWRITTEN, BUT THEY MIGHT BE USED FOR OTHER FUNCTIONS

∴ SHOULD PRESERVE VALUE OF PINS WE ARE NOT USING.

"SEE VIDEO => "LAB 0 PUZZLES CONCEPTS"

Important Background: Bitwise manipulation

Because each bit in a register can control a different pin, we will make extensive use of C's *bitwise* operators (&, |, ~) to manipulate registers.

This is a very common practice when interacting directly with hardware!

Recall the truth tables for the bitwise operators AND (&), OR (|) and NOT (~):

A	B	Z = A & B
0	0	0
0	1	0
1	0	0
1	1	1

CLEAR

X AND 0 = 0

X AND 1 = X

X	Y	Z = A | B
0	0	0
0	1	1
1	0	1
1	1	1

SET

X OR 1 => 1

X OR 0 => X

A	C = ~A
0	1
1	0

Where "X" is either 0 or 1

From these operators, **we can build a set of techniques for individually controlling specific bits in a variable while leaving the others unmodified.**

Common operations using bitwise operators

Setting individual bits to 1

We can do this by OR'ing a specific bit (or bits) with a 1. This is called "setting" a bit.

Setting individual bits to 0

We can do this by AND'ing a specific bit (or bits) with a 0. This is called "clearing" a bit.

SET BITS TO 1

V = V | 0x03

V 0101 0101

0x03 0000 0011

0101 0111

PRESERVED SET TO 1

SET BITS TO 0

V = V & 0xF0;

V 1100 1100

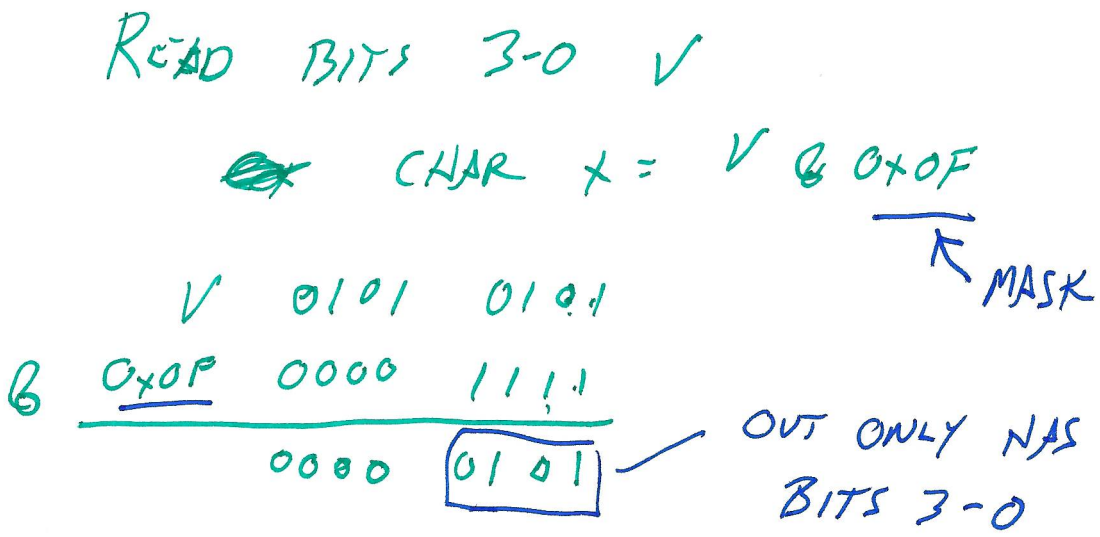
0xF0 0000 1111

0000 1100

SET TO 0 PRESERVED

"Selecting" specific bits from a variable

It is often necessary to check if certain bits of a field are set, or to only take the value of certain bits from a variable. We can do this by AND'ing a variable with only those bits that interest us set to 1–this is called *masking* bits.



You will use these techniques very frequently when working with digital I/O:

A SLIGHTLY BETTER WAY

PORT 3, PINS .7-4 OUTPUTS
PORT 3, PINS .1-0 INPUTS

P3SEL	7	6	5	4	3	2	1	0
	X	X	X	X	X	X	X	X
	0	0	0	0	1	1	0	0
	0	0	0	0	X	X	0	0

// SELECT FOR DIGITAL I/O

```
P3SEL = P3SEL & 0xGC;
```

// SET BITS 7-4 AS OUTPUTS (SET TO 1)

```
P3DIR = P3DIR | 0xF0;
```

P3DIR	X	X	X	X	X	X	X	X
	1	1	1	1	0	0	0	0
	1	1	1	1	X	X	X	X

// SET BITS 1-0 AS INPUTS (SET TO 0)

```
P3DIR = P3DIR & 0xFC;
```

P3DIR	X	X	X	X	X	X	X	X
&	1	1	1	1	1	1	0	0
	X	X	X	X	X	X	0	0

NEED TO DO THESE IN SEPARATE STEPS!

An even better way: Lose the "magic numbers"

In this lecture, it's clear what the constants 0xF0 and 0xFC mean, but will you remember what's happening here 6 months from now? Probably not.

In C, as in many programming languages, it's good practice to avoid *magic numbers*, or hard coded numbers that appear in the code without explanation of their meaning or purpose. Instead, we can use constants to attach meaning to these values and allow them to be reused.

In this case, a set of constants for the individual bits are defined for us, we can just use them:

Name	Hex	Binary	Name	Hex	Binary
BIT0	0x01	0000 0001b	BIT4	0x10	0001 0000b
BIT1	0x02	0000 0010b	BIT5	0x20	0010 0000b
BIT2	0x04	0000 0100b	BIT6	0x40	0100 0000b
BIT3	0x08	0000 1000b	BIT7	0x80	1000 0000b

We can also combine these constants to refer to more than one bit:

Ex. (BIT2|BIT1)

```
  0000 0100   BIT2
| 0000 0010   BIT1
-------------
  0000 0110
```

~(BIT2|BIT1) = ~(0000 0110) = 1111 1001

P3DIR = P3DIR & 0xF9;

P3DIR = P3DIR & ~(BIT2|BIT1);

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

6.1 6-0

"DECODER"

INPUT → LOGIC → OUTPUT

P3.1-0 OUR CODE P3.7-4

INPUT

READ P3.1-0 FROM INPUT REG

CHAR INBITS = P3IN & (BIT1 | BIT0);

OUTPUT →

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

~~P3OUT = 0;~~

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

P3OUT	X X X X
0 0 0 0	1 1 1 1
0 0 0 0	X X X X

```
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;
}
```

CONTINUING BY CREATING OUR DECODER ON OUR MSP430 BOARD...

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~~ (LAUNCHPAD BUTTONS)

Output: Given *a*, set ~~P3.7-4~~ (EXTERNAL LEDS) based on the table:

Input		Output			
P2.1	P1.1	P6.3	P6.2	P6.1	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

"DECODER"

INPUT → LOGIC → OUTPUT

~~P3.1-0~~ ~~P3.7-4~~

INPUT: LAUNCHPAD BUTTONS → A_1 = P2.1, A_0 = P1.1

OUTPUT: SOME LEDS.

↳ P6.6-3 (e.g.)

↳ HOW DO WE KNOW HOW TO CONNECT?

OUR MSP430 IS ON THE LAUNCHPAD BOARD, WHICH PROVIDES US HEADERS TO CONNECT STUFF

CONNECTING STUFF: KEY STEPS

1. FIND AVAILABLE PINS ($P_{x.y}$)
2. INPUT OR OUTPUT?
3. DETERMINE MEANING OF 1 OR 0

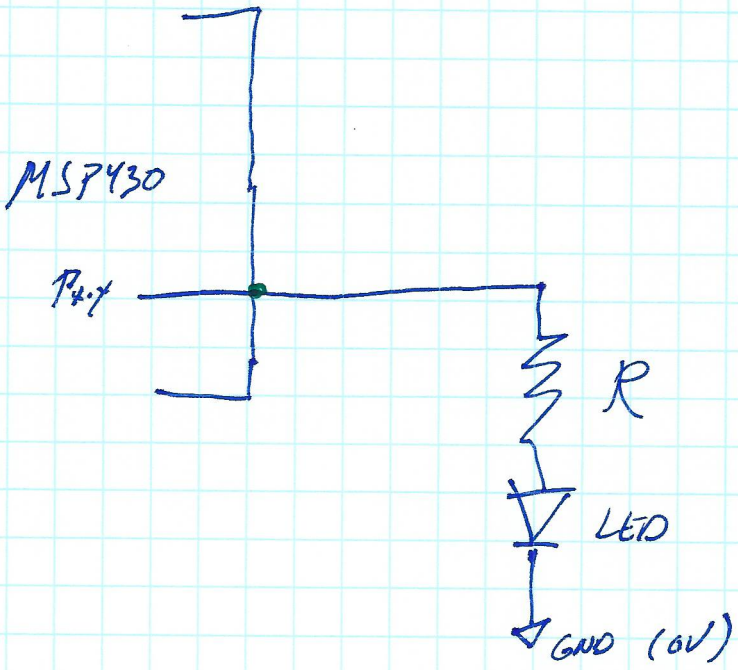
(2.5. (IF INPUT) CONFIGURE PULL UP/DOWN RESISTORS, IF REQUIRED)

SEE THE DECODER EXAMPLE FOR A COMPLETE CODE EXAMPLE ON THE COURSE WEBSITE (L6 NOTES)

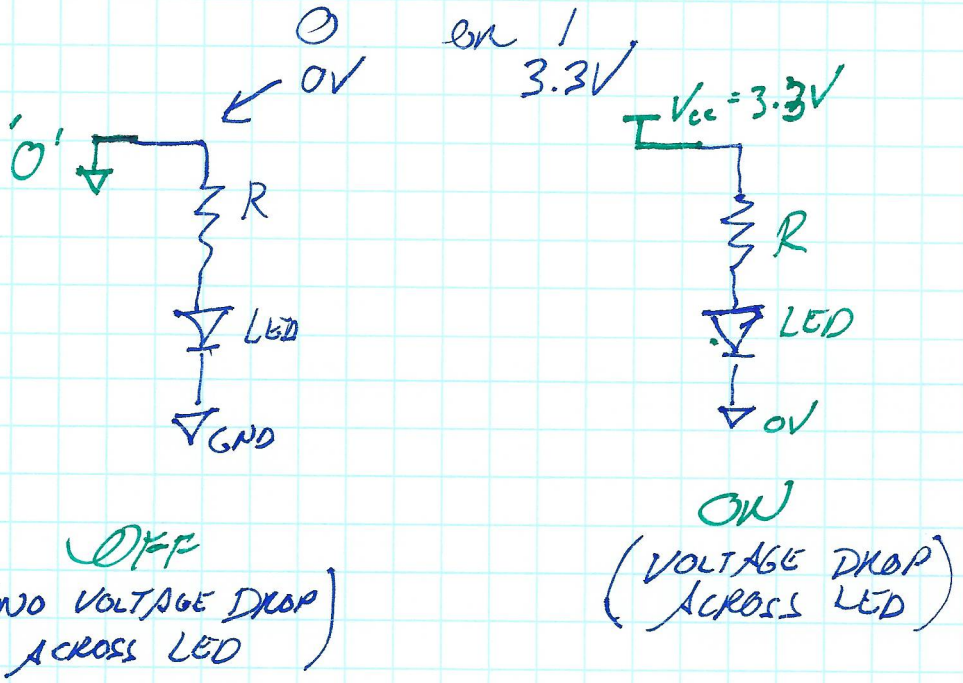
CONNECTING LEDs: A GENERIC PICTURE

AN OUTPUT

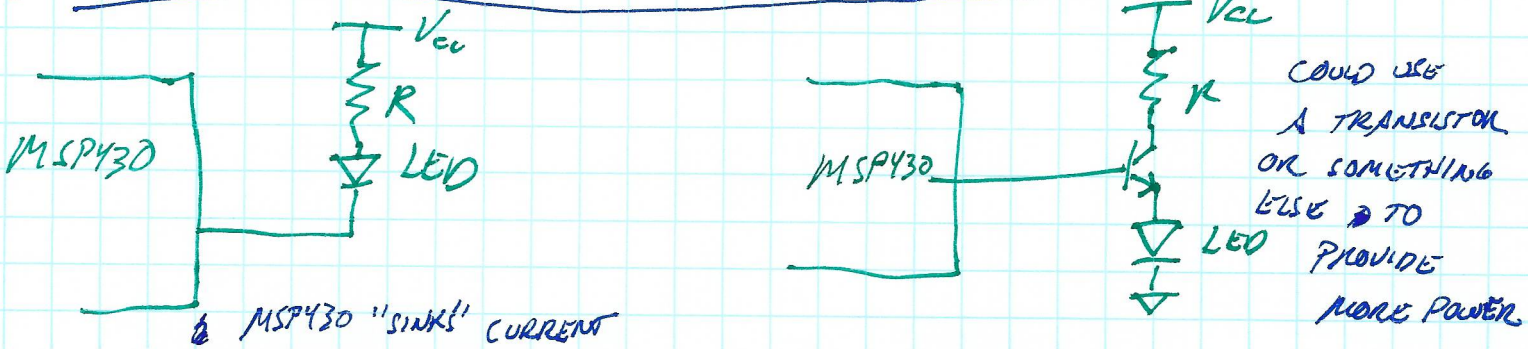
ONE WAY:



WHAT LOGIC LEVEL LIGHTS THE LED?



CAN ALSO HAVE MANY OTHER CONFIGURATIONS!

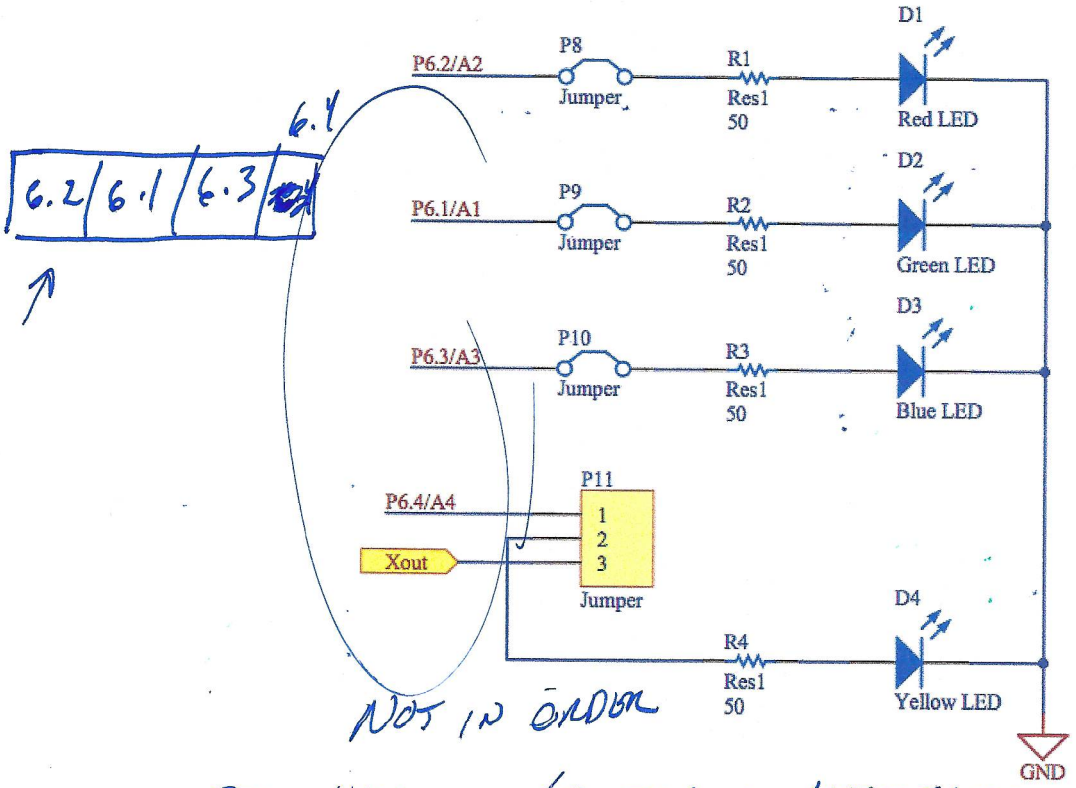


Digital I/O Concepts: Input or Output?

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**

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



6.4

6.2 | 6.1 | 6.3 | 6.4

NOT IN ORDER

HARDWARE / SOFTWARE INTERFACE:

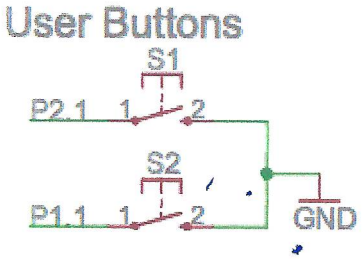
- WANT TO USE FUNCTIONALITY WITHOUT CARING ABOUT WHICH PINS PERFORM EACH FUNCTION!

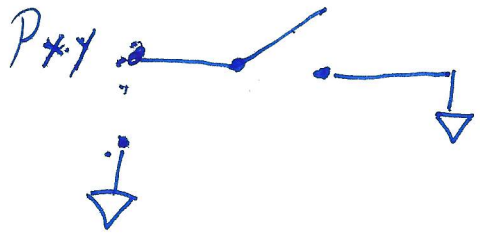
In an application program like our demo project, we use the digital I/O ports repeatedly to use the buttons and LEDs. In these programs, it's a good idea to *wrap* the functionality for hardware components into useful functions.

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

Dealing with Inputs

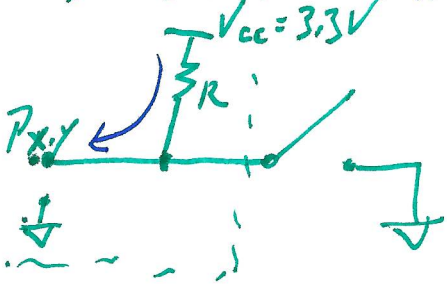
As we discussed briefly last lecture, inputs may require some special handling. Consider the buttons on the Launchpad board:





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

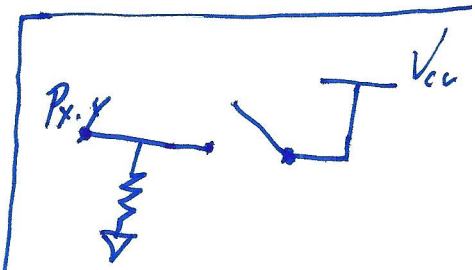


WHEN SWITCH IS OPEN => 3.3V => '1'
WHEN SWITCH IS CLOSED => 0V => '0'

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

WHY?

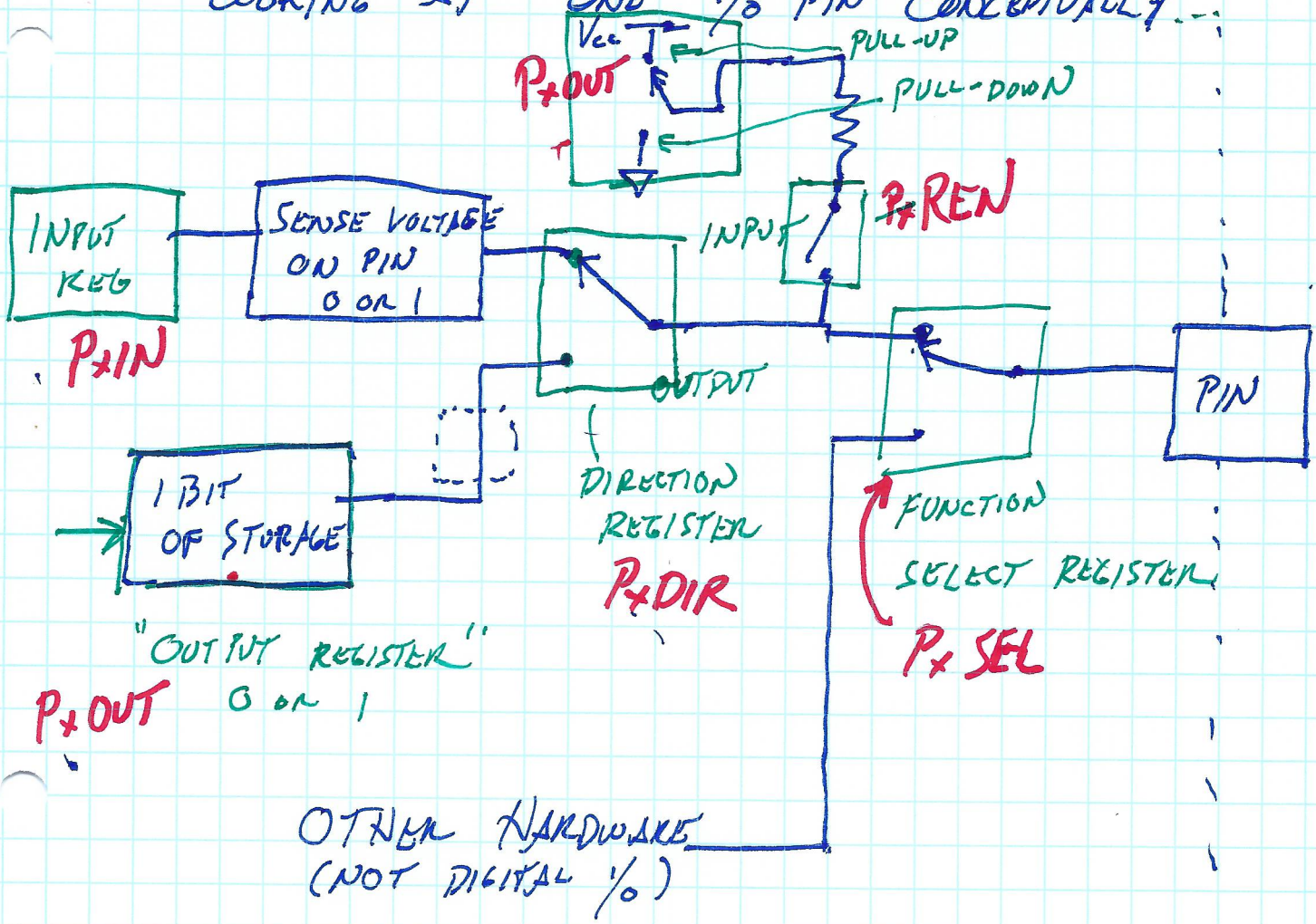
DON'T NEED TO INCLUDE PULL-UP RESISTORS ON PCB
=> LOWER COST.



CAN ALSO HAVE A PULL DOWN RESISTOR, LIKE THIS.

SO WE HAVE A SOME ADDITIONS TO THE ~~ONE~~ HARDWARE FOR THIS... (P+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/PULL 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.

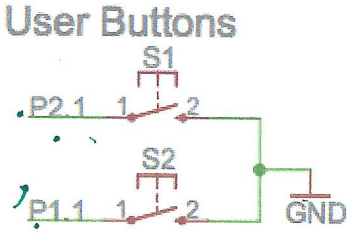
Set to 0 = Reduced drive strength (default) Set to 1 = Full drive strength

USUALLY: ≈ 10mA

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.

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
← 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)
{
                    READ LAUNCHPAD BUTTONS()




}
```

- READ P2IN → P2.1
  P1IN → P1.1