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!

Fun Facts about Digital I/O

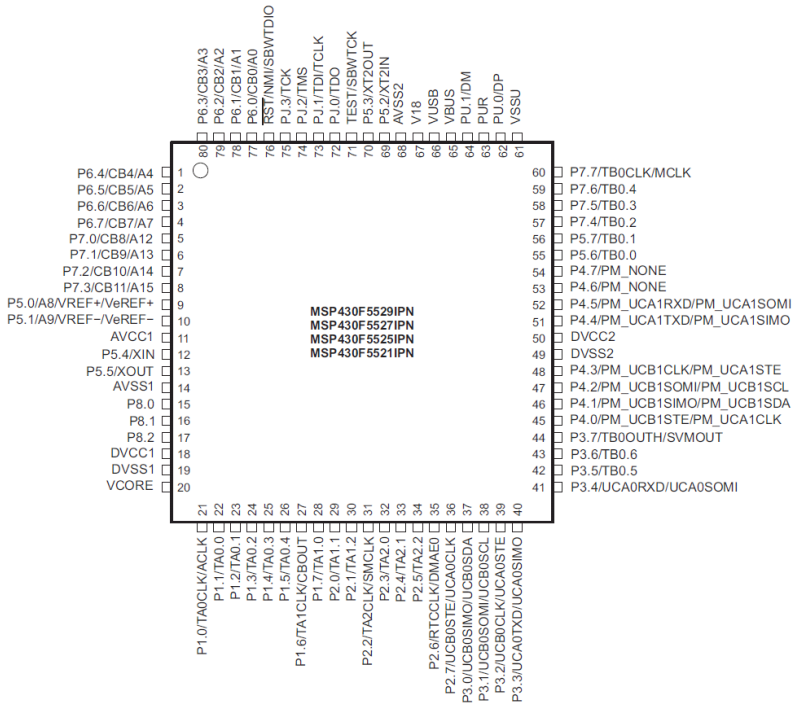
- Eight independent, individually-configurable *ports*, named P1-P8
- Ports 1-7 each have 8 configurable *pins*, and are thus 8 bits wide; Port 8 is 3 bits wide
  Pins are referenced as P<port>.<pin>, eg. P1.4.
- Each pin of each port can be configured individually as input or output
- Most digital I/O 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:

- 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*:



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

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.**

Function Select Register (PxSEL)

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

Direction Register (PxDIR)

Sets port pins as Input or Output
Set to 1 = Output
Set to 0 = Input

Input Register (PxIN)

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

Output Register (PxOUT)

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

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.

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	1	
1	0	
1	1	

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

A	C = ~A
0	
1	

Where “X” is either 0 or

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.

"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:

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:

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:

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:

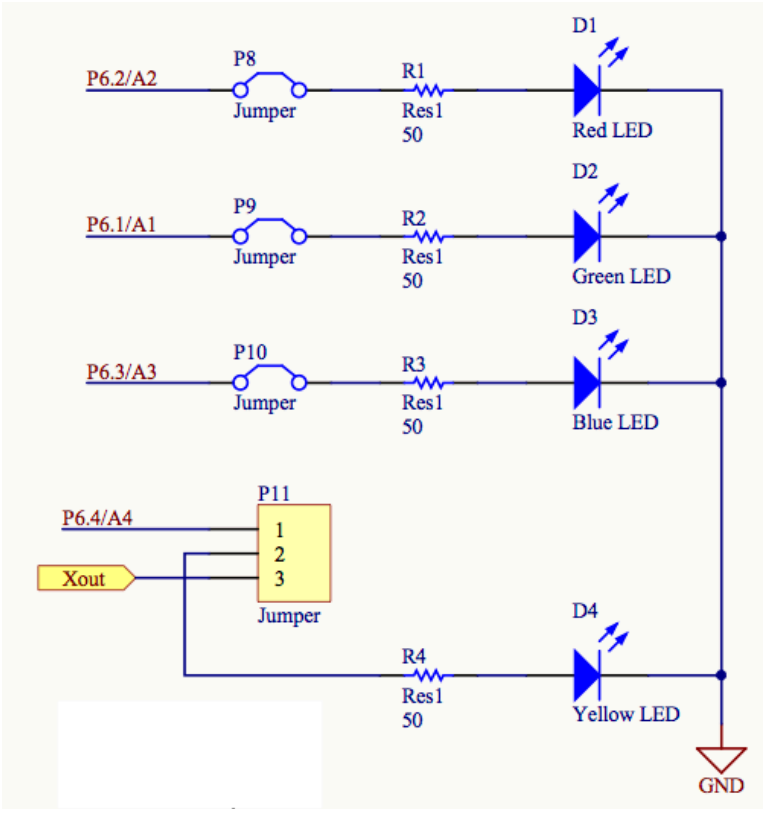
Input		**Output**			
a_1	a_0	z_3	z_2	z_1	z_0
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

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?



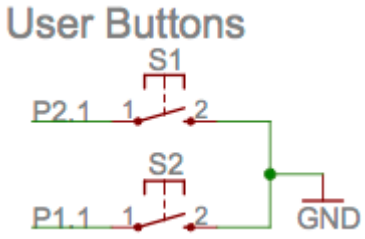
In an application program like our demo project, we use the digital I/O ports repeatedly to use the buttons and LEDs. In these programs, it's a good idea to *wrap* the functionality for hardware components into useful functions.

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:



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.

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.

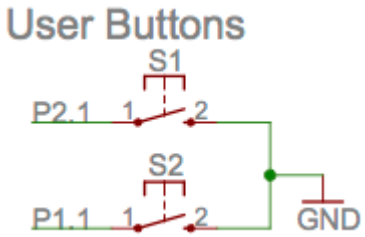
You will also see one more Digital I/O register…

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.

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;
}
```

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)
{









}
```

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?