Lab 5: Using the ATMega1284 microcontroller (2 Days)

UCR EE/CS120B

Pre-lab

Complete all Participation and challenge activities as prescribed in the Zybook assignment (Chapter 7: Laboratory Exercise #5 Material).

Be sure that you have all needed supplies, ready for use. Read over the entire lab so you'll know what to expect. You should also attempt to run the basic programs as well. The first two exercises are from previous labs so you should already have the code and test cases for them.

Wiring your board

Breadboard

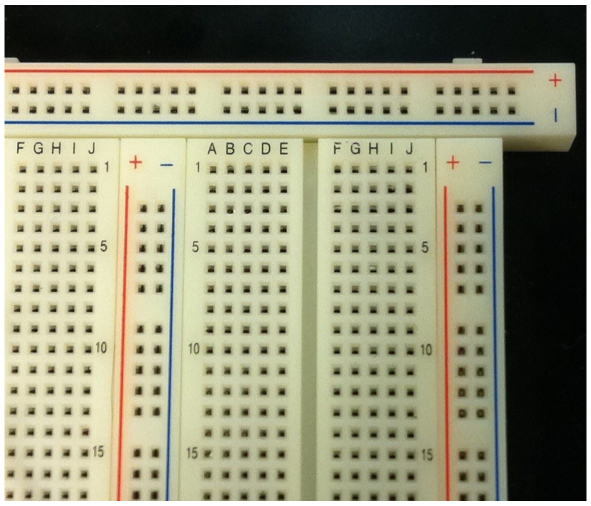
Review electronics, use of breadboards, and safety when working with electronics.

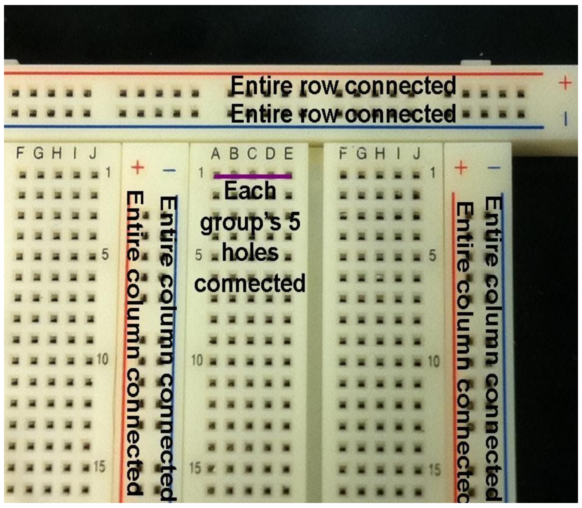
Before touching any of the sensitive circuit components, please make sure to discharge static electricity off of yourself. To properly discharge static electricity, touch a grounded metal component (i.e. metal on the lab machines). The discharge of static electricity onto a component is known as Electrostatic discharge (ESD). ESD can damage electronic components so take caution when handling sensitive electronic equipment.

Note: Always use anti-static bags and tubes as needed.

The Lab's Google Doc Collection has such helpful files such as the AVR FAQ, AVRISP user guide, ATmega1284 datasheet, and more. Some of these contain troubleshooting sections, and it is a good idea to familiarize yourself with the datasheet for the ATmega1284.

On a breadboard, the holes along a red line (also called a rail or bus) are connected internally. Likewise, the holes along a blue line are also connected internally. Red is for power, blue/black for negative (ground). For the rest of the breadboard, the only internal connections are among the holes in a 5-hole group in a row.





Basic board wiring for power

If you are using a battery pack through the regulator, follow this walkthrough first.

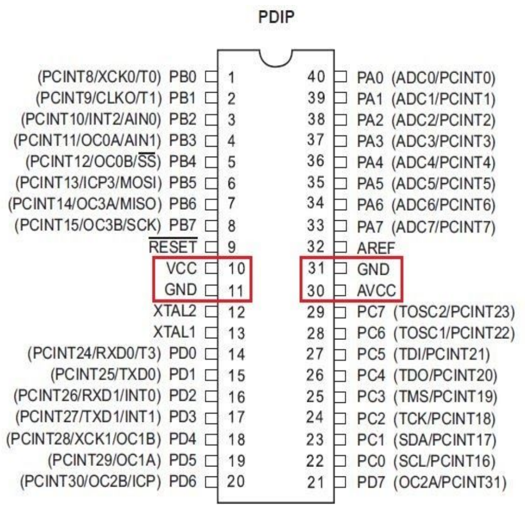
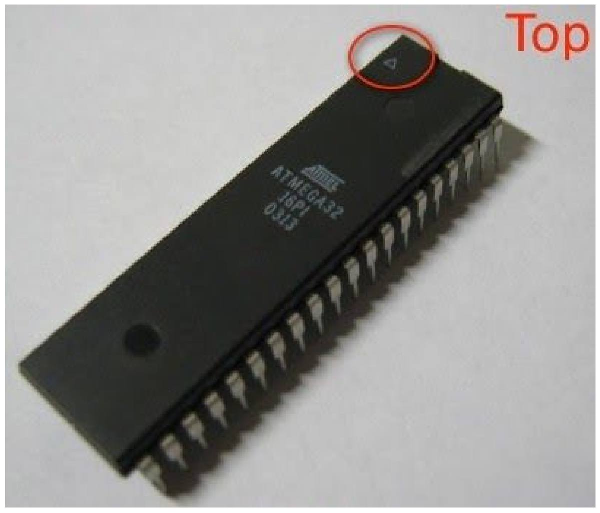
If you have a power supply, align the headers with the input pins as labelled.

Important Note: If at any time while the power supply is connected you smell burning, or feel excessive heat coming from a component *disconnect your power source immediately and check your wiring! Do not touch any components, until they cool down.* If this happens you possibly have miswired the board and are burning components. They may still work afterwards but they also may be toast! Be sure to consider this when debugging.

Add the microcontroller

*Before you begin ensure the board is **NOT** powered and you have discharged any static electricity from your person.*

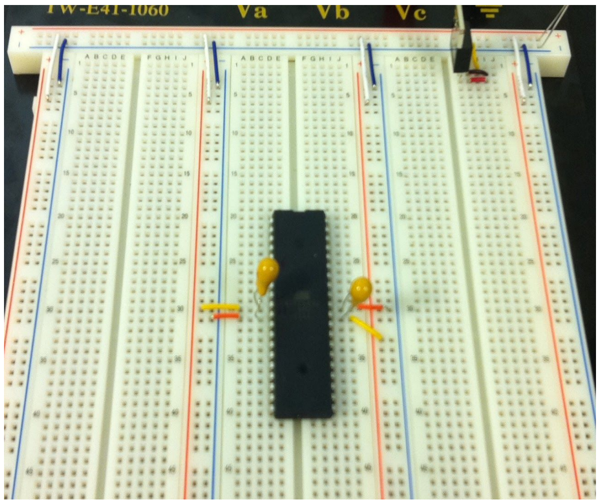
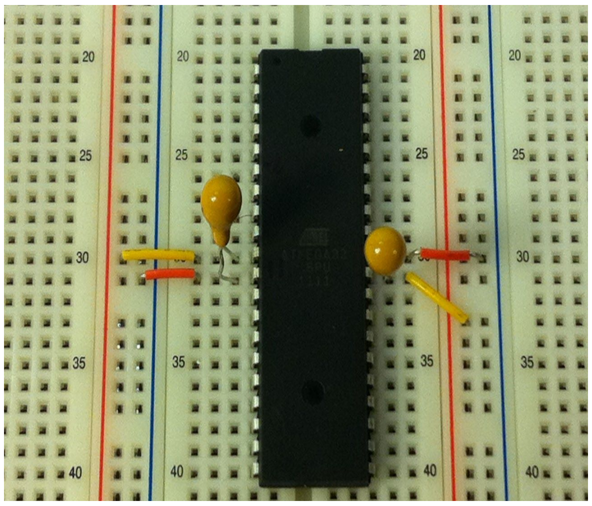
First you will want to determine the orientation of the microcontroller (sometimes abbreviated μC). There is a notch in the top of the microcontroller and pin 1 is marked with a dot or arrow as pictured below.

Carefully insert the microcontroller chip onto the breadboard as shown below. Place the top pin in row 21 to simplify determining the chip's pin numbers, such that pin *1* (upper left of chip) is in row 2*1*, pin *2* in 2*2*, pin *3* in 2*3*, etc. The chip's horizontal pin spacing is slightly wider than the board, so place one side partly in, then angle the other side's pins in carefully; once all pins are in holes, gently press downwards until the chip snaps into place. If you have to remove the chip for any reason, use the chip extractor tool -- Video demo-ing chip insertion/extraction.

Next, we will add wires for both V_{cc} and ground (gnd) to both sides of the microcontroller per the pinout above. Note that V_{cc} and gnd connect to specific pin numbers on each side as shown -- be careful to note that V_{cc} is above ground on the left, but ground is above AV_{cc} (which is V_{cc}; see datasheet) on the right.

V_{cc} (red rail) connects to both pin 10 and 30. Ground (blue rail) connects to pin 11 and 31. Add the (optional) capacitors across V_{cc} and ground, to smooth out the 5V being supplied to the microcontroller (reducing spikes or dips). This ceramic capacitor doesn't have a positive or negative side, so don't worry about its orientation (polarity). Confirm your setup with the pictures below.

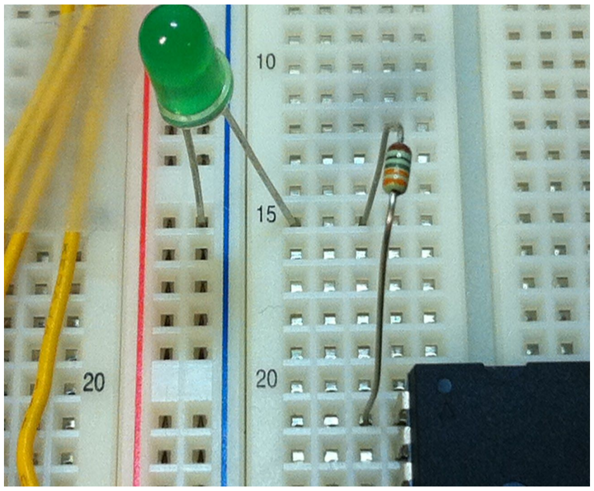
Connecting the programmer

If you don't have a header chip from IEEE follow this walkthrough. Otherwise you should be able to plug-in the header so V_{cc} and `gnd` line up (on the left hand side of the chip shown above).

A first program on the microcontroller chip

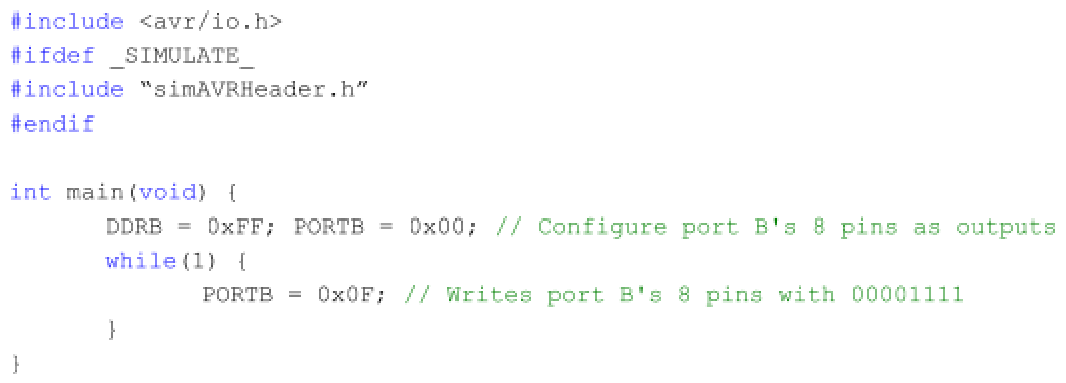
This program, from an earlier lab, just sets PB3..PB0 to 1. We'll connect an LED to PB0, so the program should turn on PB0's LED.

1. Prepare the board for the upcoming program by adding an LED to PB0.
   a. Remove power from the board (note that the AVRISP external LED turns red). It's not necessary to unplug the AVRISP.
   b. Add a 330Ω resistor and LED in series between PB0 and ground as shown. Orient the LED properly, with the negative (short) leg plugged into ground. (The resistor limits the current flow, extending the LEDs life and keeping the microcontroller cooler).



2. Create a new project (`createProject`) named "`lab_chip`" for the ATmega1284 and write the

following program in the `source/main.c` file (from an earlier lab):



```c
#include <avr/io.h>
#ifdef _SIMULATE_
#include "simAVRHeader.h"
#endif

int main(void) {
	DDRB = 0xFF; PORTB = 0x00; // Configure port B's 8 pins as outputs
	while(1) {
		PORTB = 0x0F; // Writes port B's 8 pins with 00001111
	}
}
```

3. Type `make debug` to debug the program and step through to see it executing and inspect the generated waveform (`.vcd`) or write a short test to verify it is working properly.
4. Once we've verified it works properly, we're going to download the program onto the ATmega1284 chip, a process called "programming" the chip.
5. Type `make program` and you should see output similar to below:
   Note: you can also use AVRDUDESS for a GUI interface to the programmer (`avrdudess` on the lab machines).

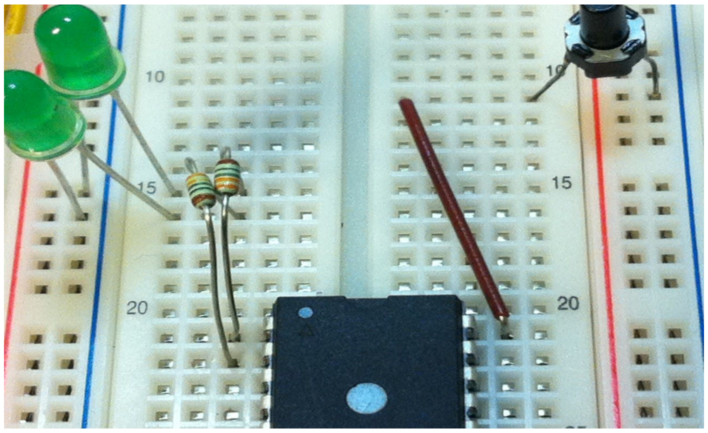
```
         Programmer Type : JTAG3_ISP
         Description     : Atmel-ICE (ARM/AVR) in ISP mode
         Vtarget         : 4.9 V
         SCK period      : 125.00 us

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e9706 (probably m1284)
avrdude: safemode: hfuse reads as D9
avrdude: safemode: efuse reads as FF
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "build/main.hex"
avrdude: input file build/main.hex auto detected as Intel Hex
avrdude: writing flash (376 bytes):

Writing | ################################################## | 100% 1.52s

avrdude: 376 bytes of flash written
avrdude: verifying flash memory against build/main.hex:
avrdude: load data flash data from input file build/main.hex:
avrdude: input file build/main.hex auto detected as Intel Hex
avrdude: input file build/main.hex contains 376 bytes
avrdude: reading on-chip flash data:
```

```
Reading | ################################################## | 100% 2.09s

avrdude: verifying ...
avrdude: 376 bytes of flash verified

avrdude: safemode: hfuse reads as D9
avrdude: safemode: efuse reads as FF
avrdude: safemode: Fuses OK (E:FF, H:D9, L:E2)

avrdude done.  Thank you.
```

A program using a button for input

This program, from an earlier lab, reads an input button and sets two LEDs to 01 (if not pressed) or 10 (if pressed).

1. Prepare the board for the upcoming program. Be sure power is removed. Add an LED to PB1, and add a button connecting to ground on one end and to PA0 on the other. When pressed, the button forms a connection; else there is no connection.

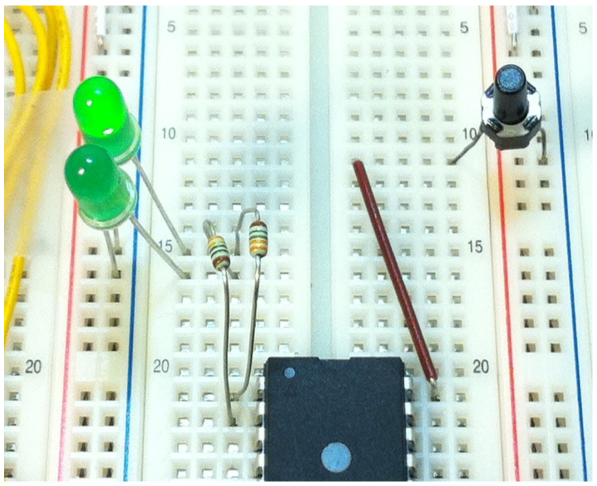
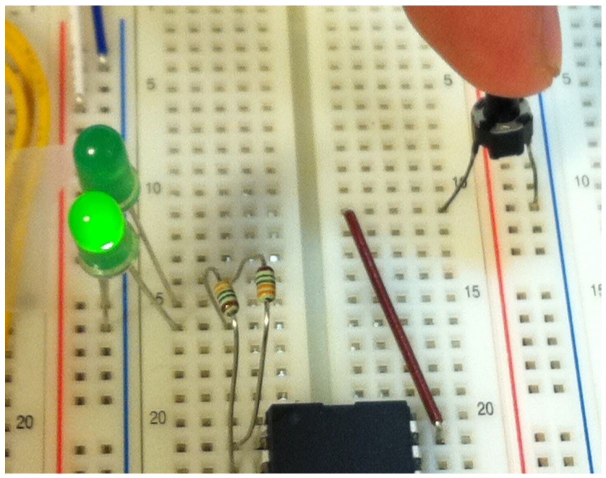
   

   Note: PA0 will be programmed in *pull-up mode*, meaning that when the pin has *no input* the program will read it as 1, and when the pin has a *0 input* (ground) the program will read it as 0. Note how the above button provides PA0 with either no input (when not pressed, so read as 1) or with 0 (when pressed, so read as 0). Therefore, ***pressing*** the button causes PA0 to be ***read as 0***, and releasing as 1.
2. Update your `source/main.c` to the following:

```c
#include <avr/io.h>
#ifdef _SIMULATE_
#include "simAVRHeader.h"
#endif

int main(void) {
        DDRA = 0x00; PORTA = 0xFF; // Configure PORTA as input, initialize to 1s
        DDRB = 0xFF; PORTB = 0x00; // Configure PORTB as outputs, initialize to 0s
        unsigned char led = 0x00;
        unsigned char button = 0x00;
        while(1) {
                // if PA0 is 1, set PB1PB0=01, else =10
                // 1) Read inputs
                button = ~PINA & 0x01; // button is connected to A0
                // 2) Perform Computation
                if (button) { // True if button is pressed
                        led = (led & 0xFC) | 0x01; // Sets B to bbbbbb01
                                                // (clear rightmost 2 bits, then set to 01)
                } else {
                        led = (led & 0xFC) | 0x02; // Sets B to bbbbbb10
                                                // (clear rightmost 2 bits, then set to 10)
                }
                // 3) Write output
                PORTB = led;
        }
}
```

3. Build the project and program the chip. (Don't forget to power the board).
   ```
   $ make program
   ```
4. PB0's LED should be on and PB1's LED off. Press the button, and note that PB0's LED turns off and PB1's LED turns on.

Note that PORTA is initialized to 0xFF -- this is essential for pull-up mode. Failing to initialize to 0xFF may result in strange errors from the port's read values being inconsistent.

Using PORTC -- Disabling JTAG required

Add 8 LEDs (with resistors) to port C's pins and write a program that sets port C to 0x00 initially and to 0xFF while the PA0 button is pressed. *Observe that, incorrectly, not all the LEDs light.* The reason is that some pins on the ATmega1284 have multiple possible purposes, as indicated in the parentheses of the chip's pinout diagram and as described in the ATmega1284 datasheet:

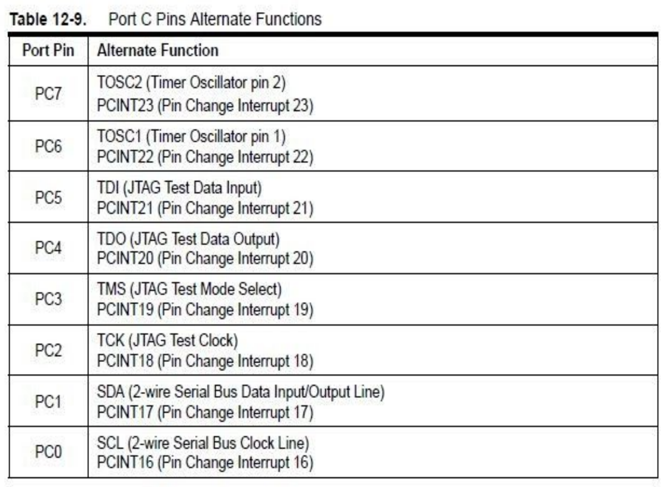
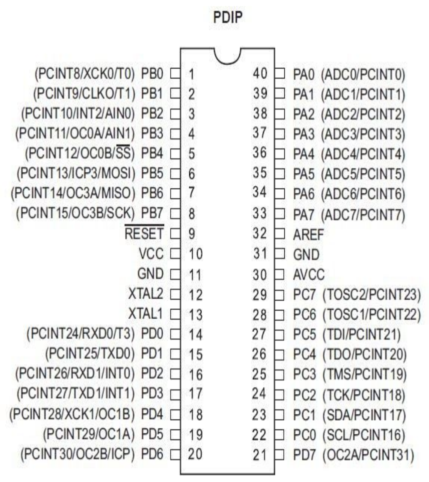


Table 12-9. Port C Pins Alternate Functions

Port Pin	Alternate Function
PC7	TOSC2 (Timer Oscillator pin 2) PCINT23 (Pin Change Interrupt 23)
PC6	TOSC1 (Timer Oscillator pin 1) PCINT22 (Pin Change Interrupt 22)
PC5	TDI (JTAG Test Data Input) PCINT21 (Pin Change Interrupt 21)
PC4	TDO (JTAG Test Data Output) PCINT20 (Pin Change Interrupt 20)
PC3	TMS (JTAG Test Mode Select) PCINT19 (Pin Change Interrupt 19)
PC2	TCK (JTAG Test Clock) PCINT18 (Pin Change Interrupt 18)
PC1	SDA (2-wire Serial Bus Data Input/Output Line) PCINT17 (Pin Change Interrupt 17)
PC0	SCL (2-wire Serial Bus Clock Line) PCINT16 (Pin Change Interrupt 16)

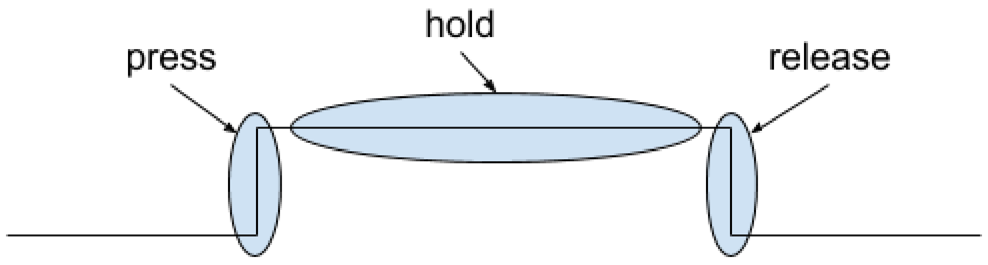


"JTAG" is a standard serial interface for advanced chip testing, which we won't be using. Fuses internal to the chip determine which purpose is active for each port. The JTAGEN fuse is bit 6 on the HIGH fuse on the ATMega1284 (see fuses calculator). To disable this fuse (while leaving other defaults on):

```
$ avrdude -c atmelice_isp -p atmega1284 -U hfuse:w:0xD9:m
```

Exercises

Write C programs for the following exercises for an ATmega1284 following the PES *standard technique*. These are similar to previous exercises you have run in the simulator. For any behavior response caused by a button **press**, the response should occur almost immediately upon the press, not waiting for the button **release** (unless otherwise stated). Be sure to count each button press only once, no matter the duration the button is pressed.



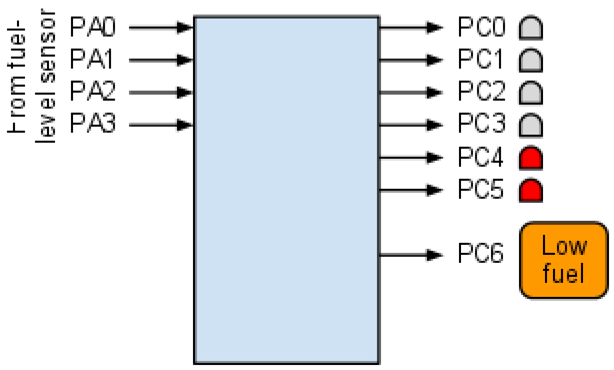
When completing these exercises, chances are that a few bugs will be encountered along the way. LEDs are a helpful component for debugging hardware.

LEDs can be used to check the input/output of individual pins. For example, If a specific pin is meant to be a certain value in order for your program to proceed, connect an LED to the pin to ensure the pin is the correct value.

A helpful hint is to always keep an extra LED connected between ground and a long wire. Whenever you wish to check the value of a desired pin, just connect the other end of the long wire to the desired pin and observe the behavior of the LED.

It is also recommended that you review the AVR FAQ to help you with common debugging issues when working with hardware.

1. (From an earlier lab) A car has a fuel-level sensor that sets PA3..PA0 to a value between 0 (empty) and 15 (full). A series of LEDs connected to PC5..PC0 should light to graphically indicate the fuel level. If the fuel level is 1 or 2, PC5 lights. If the level is 3 or 4, PC5 and PC4 light. 5-6 lights PC5..PC3. 7-9 lights PC5..PC2. 10-12 lights PC5..C1. 13-15 lights PC5..PC0. Also, PC6 connects to a "Low fuel" icon, which should light if the level is 4 or less. Use buttons on PA3..PA0 and mimic the fuel-level sensor with presses.



Video Demonstration:
http://youthttp://www.youtube.com/watch?v=_6BjqDXpdVk&feature=youtu.behttp://www.youtube.com/watch?v=_6BjqDXpdVk&feature=youtu.beu.be/_6BjqDXpdVk

2. (From an earlier lab) Buttons are connected to PA0 and PA1. Output for PORTC is initially 0. Pressing PA0 increments PORTC (stopping at 9). Pressing PA1 decrements PORTC (stopping at 0). If both buttons are depressed (even if not initially simultaneously), PORTC resets to 0. If a reset occurs, both buttons should be fully released before additional increments or decrements are allowed to happen. Use LEDs (and resistors) on PORTC. Use a state machine (*not* synchronous) captured in C.

   Note: Make sure that one button press causes only one increment or decrement respectively. Pressing and holding a button should **NOT** continually increment or decrement the counter.

Video Demonstration: http://youtu.be/a0cX_cjr5t0

3. (**Challenge**) Create your own festive lights display with 6 LEDs connected to port PB5..PB0, lighting in some attractive sequence. Pressing the button on PA0 changes the lights to the next configuration in the sequence. Use a state machine (not synchronous) captured in C.

Video Demonstration: http://youtu.be/ceOSKTxOP74

Submission

Each student must submit their source files (`.c`) and test files (`.gdb`) through Gradescope according to instructions in the lab submission guidelines.

Don't forget to commit and push to Github before you logout!