UCR EE/CS 120B

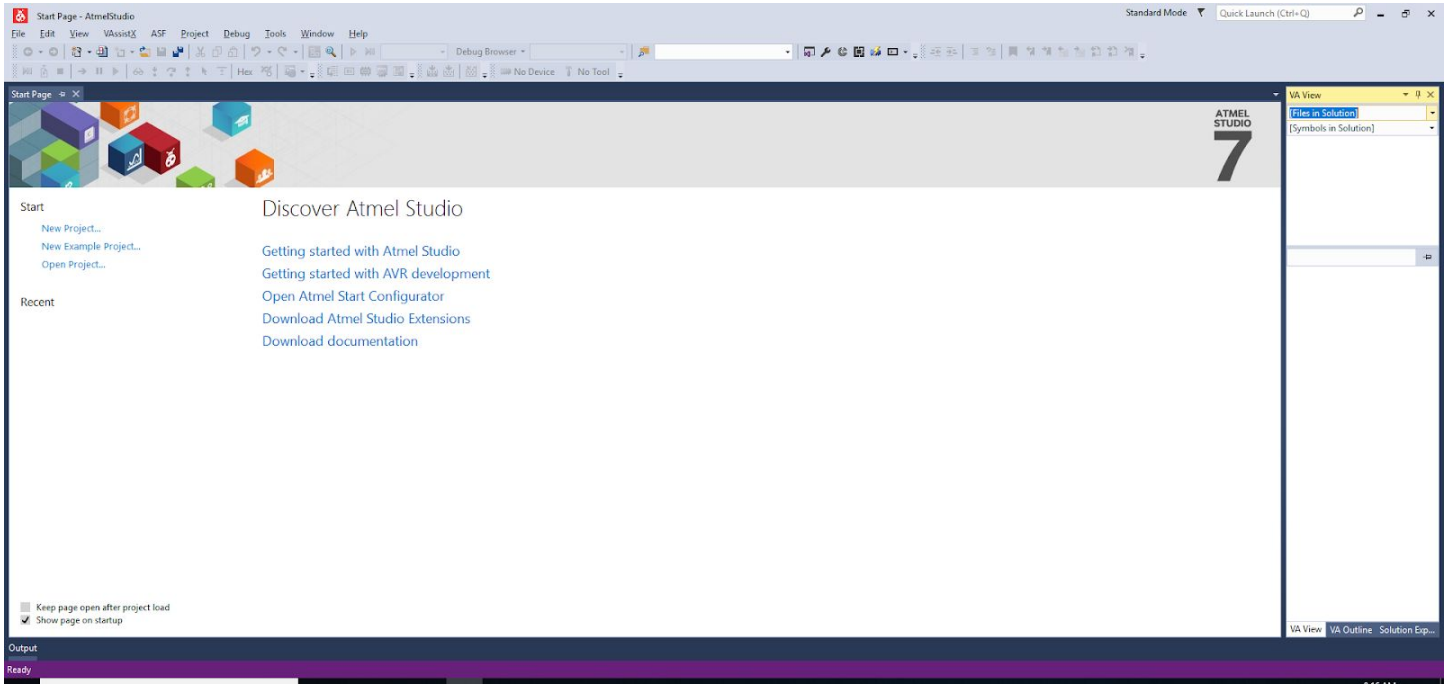
Lab 2: Intro to Atmel Studio 7 software

AVR microcontrollers are programmed using software called Atmel Studio 7 . This lab introduces Atmel Studio 7 , including the C editor, compiler, simulator, and debugger functionalities. To open Atmel Studio 7 on a lab machine, first login and open a terminal. Type "win10" to start a Windows virtual machine.

Before you start Atmel Studio, you need to create a Github account for yourself if you do not already have one. You also need to download and install Github Desktop. Then, follow this walkthrough to connect Github Desktop to your Github account. Then open Atmel and follow the below steps to start your first project.

Important: NEVER close VM by pressing the "X" in the top right corner of the window. Doing so leaves the process running in the background, preventing other users from starting a new virtual machine. **Instead**, shut down VM by going to start->shutdown from the start menu of Windows.

Important: **VM is a static image and will not save local files when the virtual machine is rebooted, thus causing all of your work to be lost.** It is important to save your work elsewhere. There is a github client in the VM that you can use to sync your work. It is recommended that you create a single CS120B-Labs repository with a folder for each lab. This way there is only one set of files to sync between the VM and your home machine. The walkthrough will help you to do this.



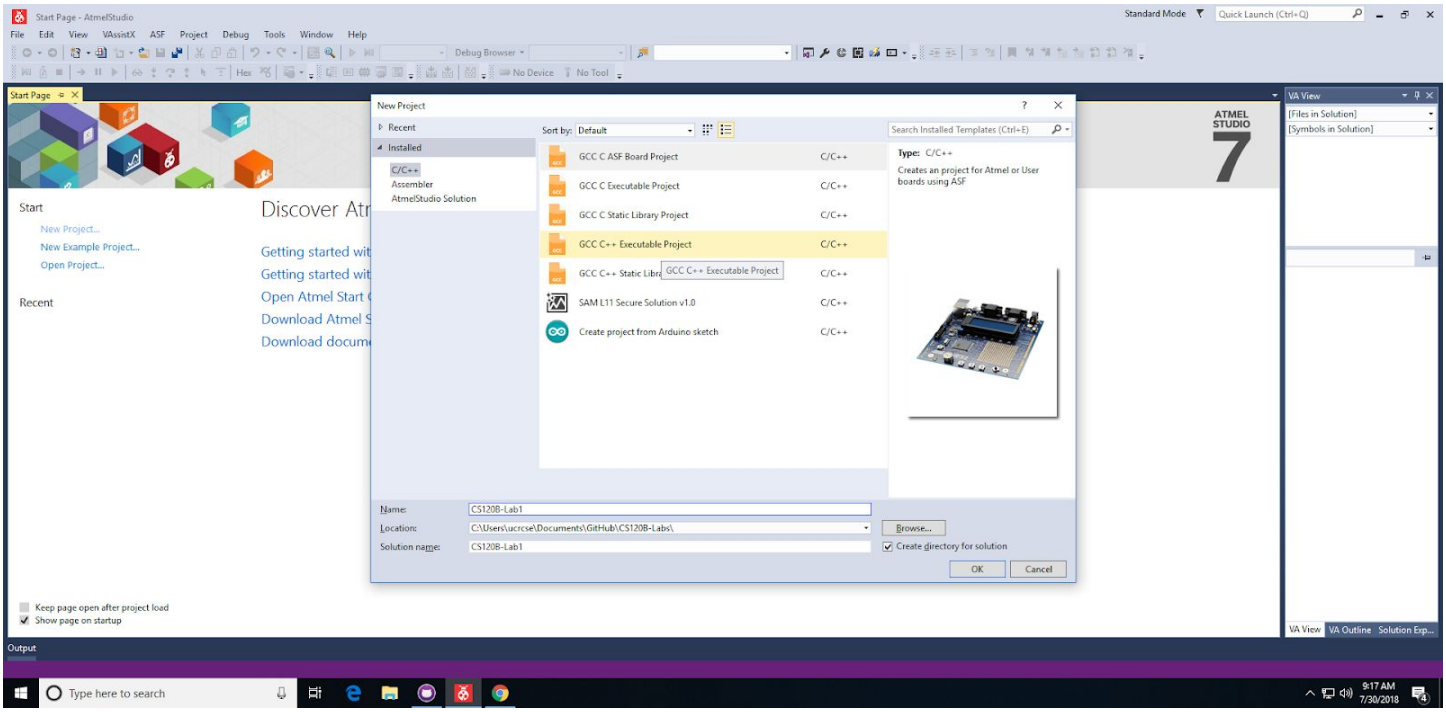
Using spaces instead of tabs.

Tabs are not standardized across all machines, or even all text editors. This will allow you to insert spaces instead of tabs to make what you write more portable.

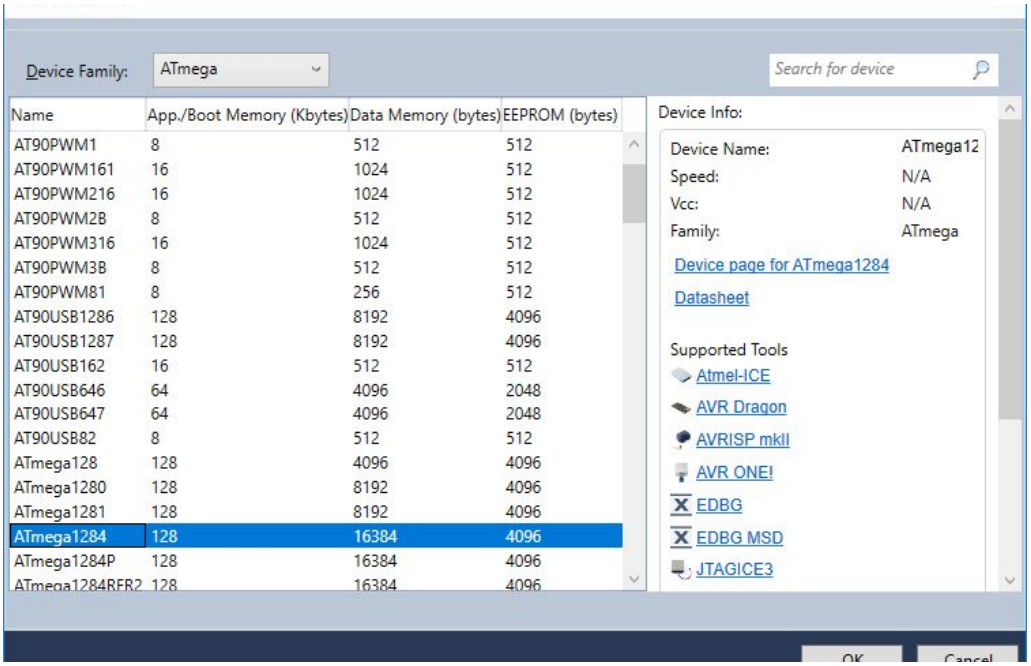
1. Open Atmel Studio 7 from the Start Menu or the Desktop.
2. Select "Tools -> Customize -> Commands -> Keyboard -> Text Editor -> Plain Text -> Tabs" (from the menu at the top).
3. Select “Insert Spaces”

Creating, compiling, and simulating a new project

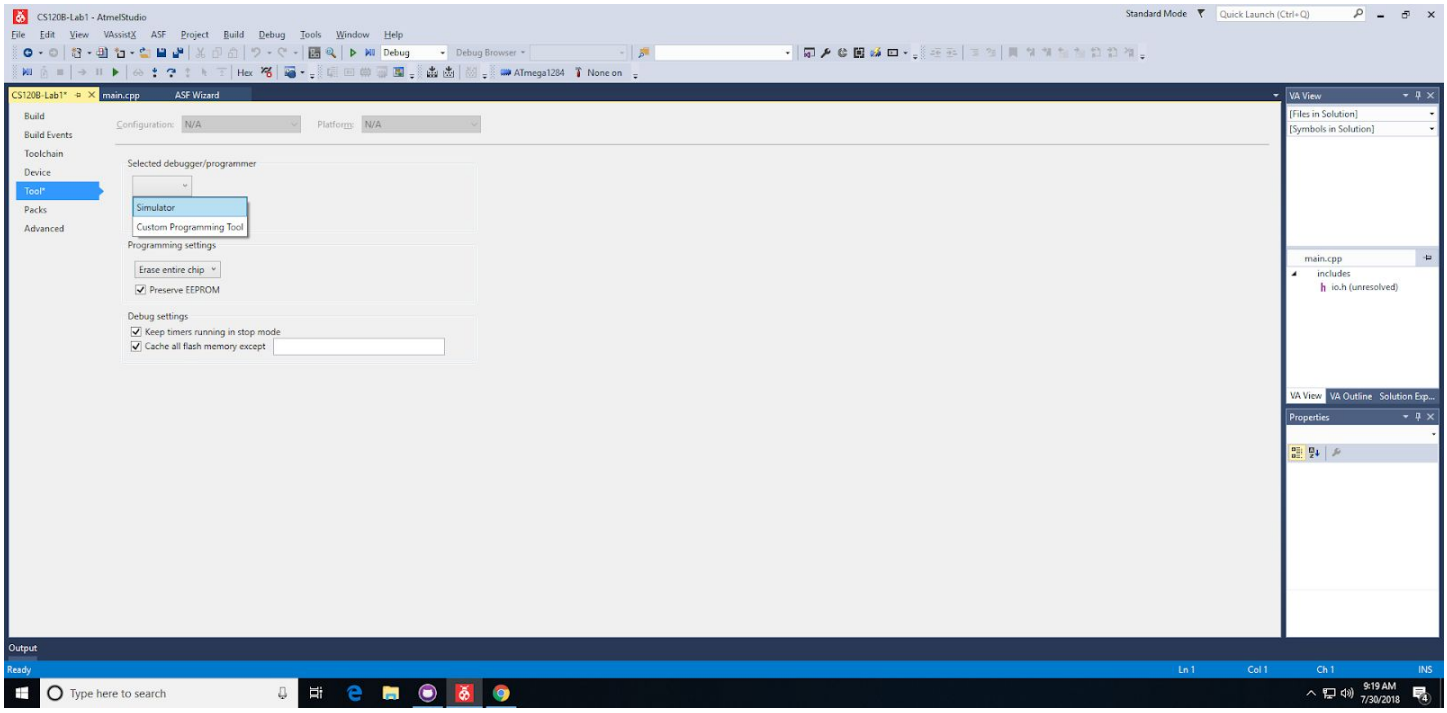
4. Open Atmel Studio 7 from the Start Menu or the Desktop.
5. Select "File -> New -> Project" (from the menu at the top).
6. Select "GCC C Executable Project" from the New Project window that pops up, type "[cslogin]_lab[#]_part[#]" in the name field near the bottom, filling in the appropriate values for this particular lab. **Don’t forget to change the location to the location of your Github folder.** Then press "OK".



7. Select "ATmega1284" from the list in the Device Selection window that pops up, press "OK"



8. A sample program appears with an empty "while(1)" loop. Select "Build -> Build solution".
9. Select "Debug -> Continue". Select the “AVR Simulator” debugger and choose “OK”. The sample program is now running in the simulator, though it has no useful behavior so there's nothing to see. Select "Debug -> Stop debugging". (If a ‘No Source Available’ error occurs, ignore it and click the ‘xxx.c’ tab at the top and press continue again.)



10. Select “Project”, then “<project_name> Properties...”. A new tab should open. Click the new tab then “Toolchain” -> “Optimization”. Change the “Optimization Level” to “None -O0”.

First program: Writing to output pins

1. Replace the sample program by the following program (explained below) that sets port B's 8 pins to 00001111:

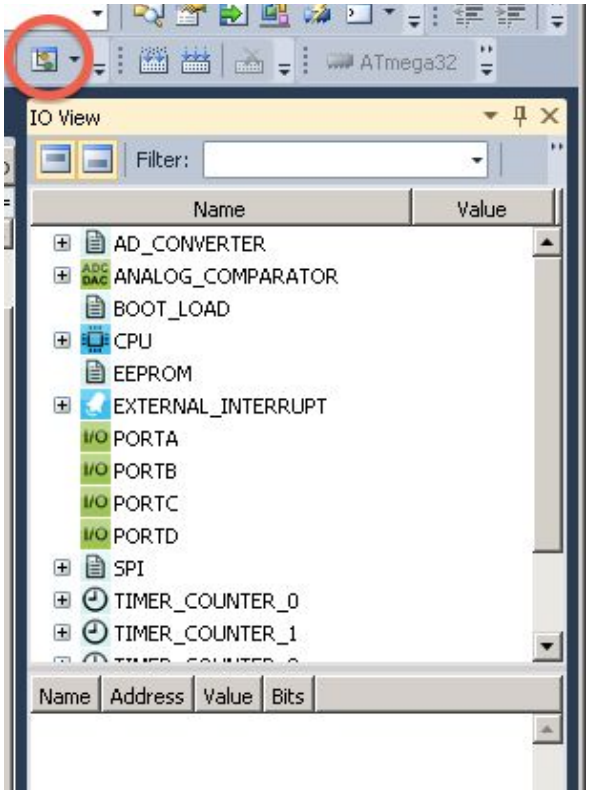
```
#include <avr/io.h>

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

2. Select "Build -> Build solution". Check the Output window to ensure the build succeeded without any errors.

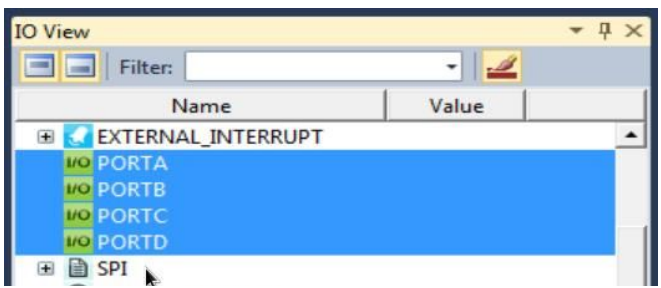
```
Build succeeded.
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

3. Select "Debug -> Continue".
4. Find and press the "I/O View" button (near the top, which has a symbol with a couple filled circles and a line), causing the "IO View" sub-window to appear.

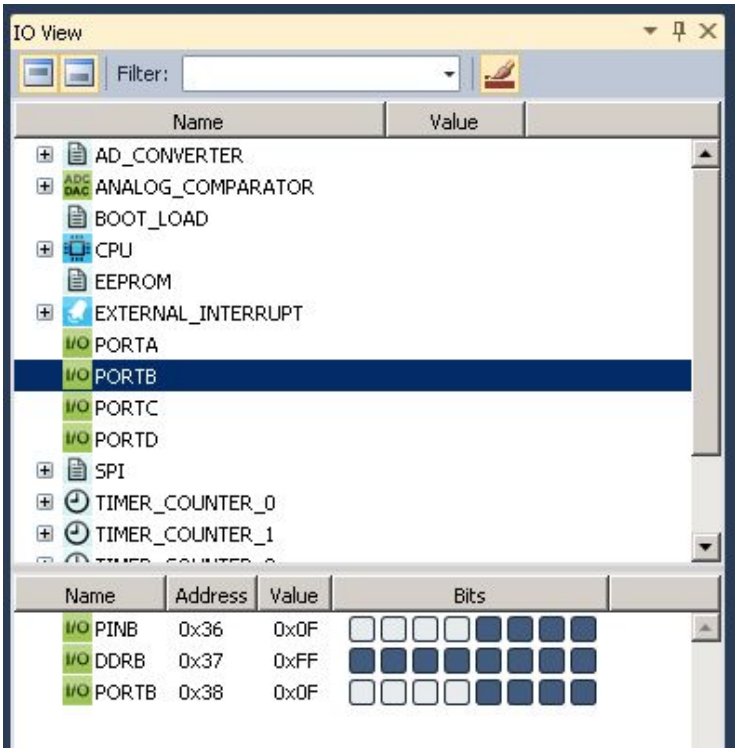


5. Select "Debug -> Break all". This pauses the program and enables viewing of the current values on the ports.

6. Hold CTRL then left-click on each PORT. All ports will now show at once. This makes for easier debug/demo as you can modify all ports/pins at once.



7. For now focus on PORTB. Click on "PORTB" in the IO View window. Notice that PORTB (further down) shows 00001111 (four unfilled dots and four color-filled dots -- you may need to adjust your view size).



8. Select "Debug --> Stop debugging"

The ATmega1284 has 4 8-bit ports named A, B, C, and D, each port being 8 physical pins on the chip. Each port has a corresponding 8-bit register DDRA, DDRB, DDRC, and DDRD (Data Direction Registers) that configure each port's pins to either an input (0) or an output (1). Thus, for example "DDRB = 0xFF;" configures all 8 pins of port B to be outputs; “DDRB = 0xF0;” configures the high nibble to output, and the lower nibble to input.. Each port also has a corresponding 8-bit registers PORTA, PORTB, PORTC, and PORTD for writing to the port's physical pins.

Note: Menu options, windows, and panels may change, appear or disappear depending on which mode you are in (e.g. debug).

Note: The above "Select" commands each have a graphic button: File->Save, a disk icon; Debug->Continue, a green triangle icon (like a play button); etc. on the toolbar. Some also can be selected using function keys, e.g., F7 causes Build Solution. Using buttons or function keys saves time.

Icon	Name	Shortcut
	Start Debugging and Break	Alt+F5
	Stop Debugging	Ctrl+Shift+F5
	Start Without Debugging	
	Continue	F5
	Restart	Ctrl+F5
	Break All	
	Step Into	F11
	Step Over	F10
	Reset	Shift+F5
	Save	Ctrl+S
	IO View	

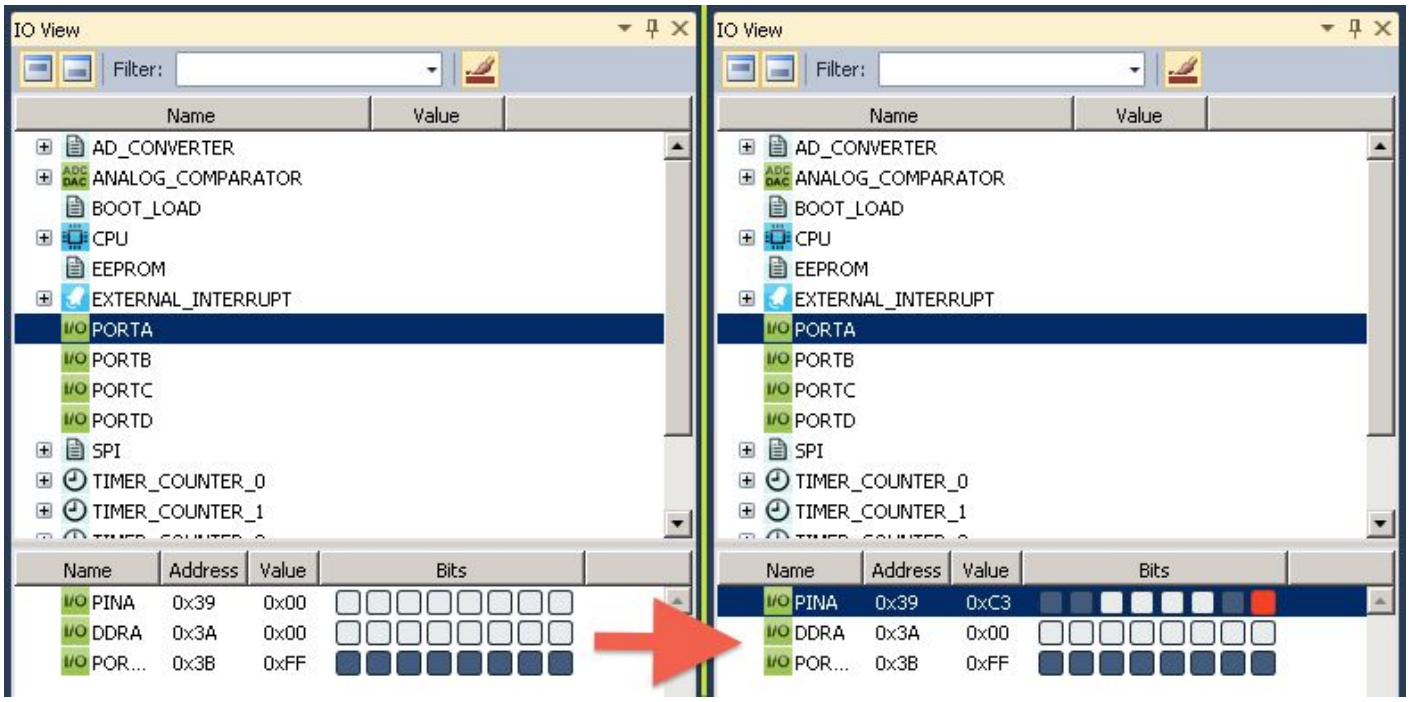
Reading from input pins

1. Modify the program into the following program:

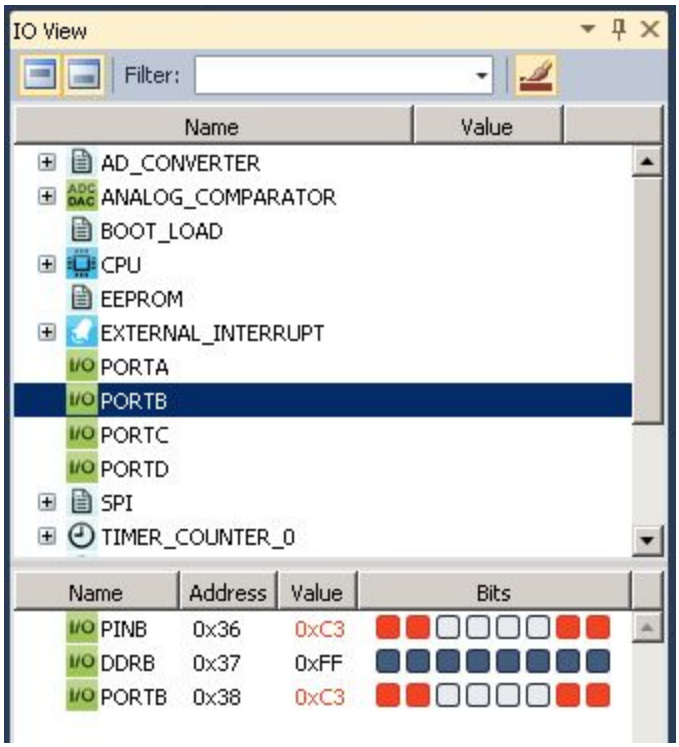
```
#include <avr/io.h>

int main(void)
{
	DDRA = 0x00; PORTA = 0xFF; // Configure port A's 8 pins as inputs
	DDRB = 0xFF; PORTB = 0x00; // Configure port B's 8 pins as outputs
	                           // Initialize output on PORTB to 0x00
	unsigned char temp_input = 0x00;
	while(1)
	{
		temp_input = PINA;
		PORTB = temp_input; // Writes port B's 8 pins with the values
		                    // on port A's 8 pins
	}
}
```

2. Select "Build -> Build solution". Ensure the build succeeded without any errors.
3. Select "Debug -> Continue".
4. Select "Debug -> Break all".
5. In the "IO View" sub-window, select PORTA. Below appears PINA, DDRA, and PORTA values. Click on the PINA bit squares to set PINA to 11000011 (two filled dots, four unfilled, two filled) within the simulator.



6. Select "Debug -> Continue". The sample program is now executing in the simulator.
7. Select "Debug -> Break all".
8. Click on "PORTB" in the IO View window. Notice that PORTB has a value of 11000011 (0xC3).



9. You can change PINA again, continue, break all, and view port B again and note that B should match A.

10. Select "Debug --> Stop debugging"

Each port has yet another corresponding 8-bit register PINA, PINB, PINC, and PIND for reading the values of the port's pins. However, for electrical reasons, the program must first write 1s to a pin (just once, at the beginning of a program) before reading, after which an external device can set the pin to 0 or 1. In summary, the three corresponding 8-bit registers for a port, say port A, are:

- DDRA: Configures each of port A's physical pins to input (0) or output (1)
- PORTA: Writing to this register writes the port's physical pins **(Write only)**
- PINA: Reading this register reads the values of the port's physical pins **(Read only)**

Note: A common AVR programming mistake is to read the PORTA register rather than reading PINA.
A common AVR Studio mistake is to try to set the PINA bits in the I/O View window without first breaking -- the bits cannot be set unless break has been selected first. Another common AVR Studio mistake is to click on the PORTA bits rather than PINA when trying to set input values -- when simulating external inputs, you set the PINA values (whereas the C program sets the PORTA values).

Always strive to read from input (PINx) and write to output (PORTx).
Mixing these up may cause odd behavior.

Accessing individual pins of a port

1. Modify the program into the following program:

```
#include <avr/io.h>

int main(void)
{
	DDRA = 0x00; PORTA = 0xFF; // Configure port A's 8 pins as inputs
	DDRB = 0xFF; PORTB = 0x00; // Configure port B's 8 pins as outputs, initialize to 0s
	unsigned char tmpB = 0x00; // Temporary variable to hold the value of B
	unsigned char tmpA = 0x00; // Temporary variable to hold the value of A
	while(1)
	{
		// 1) Read input
		tempA = PINA & 0x01;
		// 2) Perform computation
		// if PA0 is 1, set PB1PB0 = 01, else = 10
		if (tmpA == 0x01) { // True if PA0 is 1
			tmpB = (tmpB & 0xFC) | 0x01; // Sets tmpB to bbbbbb01
						     // (clear rightmost 2 bits, then set to 01)
		}
		else {
			tmpB = (tmpB & 0xFC) | 0x02; // Sets tmpB to bbbbbb10
						     // (clear rightmost 2 bits, then set to 10)
		}
		// 3) Write output
		PORTB = tmpB;
	}
	return 0;
}
```

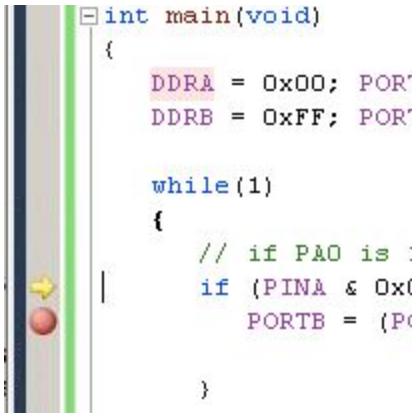
2. **NOTE:** Use of the notation "PA0", "PB1", and "PB0" in the comments to refer to port A's bit 0, port B's bit 1, and port B's bit 0, respectively. We will use such notation extensively in comments and lab assignment text, but realize that those are NOT recognized identifiers by the AVR C compiler.
3. **NOTE:** Use of a temporary variable tmpB instead of reading from PORTB.
4. Build and run the program as before. In the IO View, set PA0 to 0, observe port B. Set PA0 to 1, observe port B. Don't forget to that you must "Break all" to set or observe port values.

The code shows a common method for reading one pin of a port: "PINA & 0x01" to read PA0 (as another

example, "PINA & 0x08" would read PA3, resulting in 0x00 if PA3 was 0 or 0x08 if PA3 was 1). The code also shows a common method for writing to a particular pin (or pins) of a port: "PORTB = (PORTB & 0xFC) | 0x01" first clears PB1 and PB0, then writes "01" to those pins. Note that the selected masks preserve the other bits of port B.

Using the debugger

- Stepping
  1. Run the above program again.
  2. Select "Debug -> Break all" as you've done before.
  3. Select "Debug -> Step into", causing execution of one C statement. Notice the arrow next to the C code indicating the current statement. Select "Debug -> Step into" several more times.
  4. Set PINA.0 to 1 or 0 and use such stepping to observe the program flow.
- Breakpoints
  1. (Continuing the above...)
  2. Set a breakpoint in the above program by clicking to the far left of the "if" statement (left of the green bar). Notice that a red circle appears.
  3. Select "Debug -> Continue". Notice that the program runs briefly, then breaks where you set the breakpoint. Select "Debug -> Continue" again, and notice the program runs briefly until it again reaches the breakpoint.



Pre-lab

Read the above and write the main C code for exercises 1 and 2 targeting AVR Studio.

Exercises

For each exercise, create a new project named [cslogin]_lab1_part1, [cslogin]_lab1_part2, etc. Each part should be demoed to a TA. If practical, demonstrate a working part to your TA before moving on to the next exercise (if students are waiting, you can move on, but demo as soon as possible). "Challenge" exercises are to be completed if time permits.

1. Garage open at night-- A garage door sensor connects to PA0 (1 means door open), and a light sensor connects to PA1 (1 means light is sensed). Write a program that illuminates an LED connected to PB0 (1 means illuminate) if the garage door is open at night.

PA0 = garage door sensor (input), PA1 = light sensor (input), PB0 = LED (output)

Input	Input	Output
PA1	PA0	PB0
0	0	0

0	1	1
1	0	0
1	1	0

2. Port A's pins 3 to 0, each connect to a parking space sensor, 1 meaning a car is parked in the space, of a four-space parking lot. Write a program that outputs in binary on port C the number of available spaces (Hint: declare a variable "unsigned char cntavail"; you can assign a number to a port as follows: `PORTC = cntavail;`).

3. Extend the previous program to still write the available spaces number, but only to PC3..PC0, **and** to set PC7 to 1 if the lot is full.

4. (**Challenge**) An amusement park kid ride cart has three seats, with 8-bit weight sensors connected to ports A, B, and C (measuring from 0-255 kilograms). Set PD0 to 1 if the cart's total passenger weight exceeds the maximum of 140 kg. Also, the cart must be balanced: Set port PD1 to 1 if the difference between A and C exceeds 80 kg. Can you also devise a way to inform the ride operator of the approximate weight using the remaining bits on D? (Interesting note: Disneyland recently redid their "It's a Small World" ride because the average passenger weight has increased over the years, causing more boats to get stuck on the bottom).

   (Hint: Use two intermediate variables to keep track of weight, one of the actual value and another being the shifted weight. Binary shift right by one is the same as dividing by two and binary shift left by one is the same as multiplying by two.)

Each student must submit their .c source files according to instructions in the lab submission guidelines.