

EE/CS 120B: Introduction to Embedded Systems
University of California, Riverside
Winter 2016

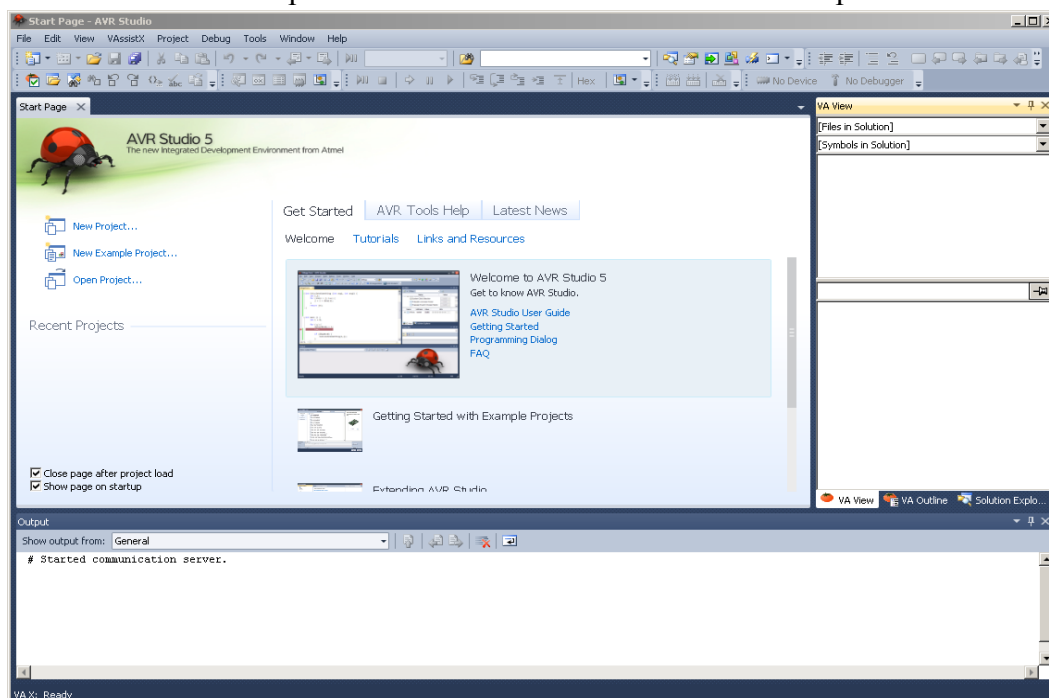
Laboratory Exercise 1

AVR microcontrollers are programmed using software called AVR Studio 6 (Recently renamed to Atmel Studio). This lab introduces AVR Studio 6, including the C editor, compiler, simulator, and debugger functionalities. To open AVR Studio 6 on a lab machine, first login and open a terminal. Type "win7" to start a Windows virtual machine; AVR Studio 6 should be on the new virtual desktop. If you want to run Atmel studio on your laptop or home PC, you can download it at the following URL: <http://www.atmel.com/tools/atmelstudio.aspx>

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 the VM by going to start->shutdown from the start menu of Windows.

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. Either to a mapped network drive (your CS home directory), a USB flash drive, or email your files to yourself. Instructions for how to map a network drive can be found in the Read Me!.txt file on the VM Windows Desktop. Additionally you can drag and drop files between the Linux desktop and the Windows virtual machine desktop.



Use 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.

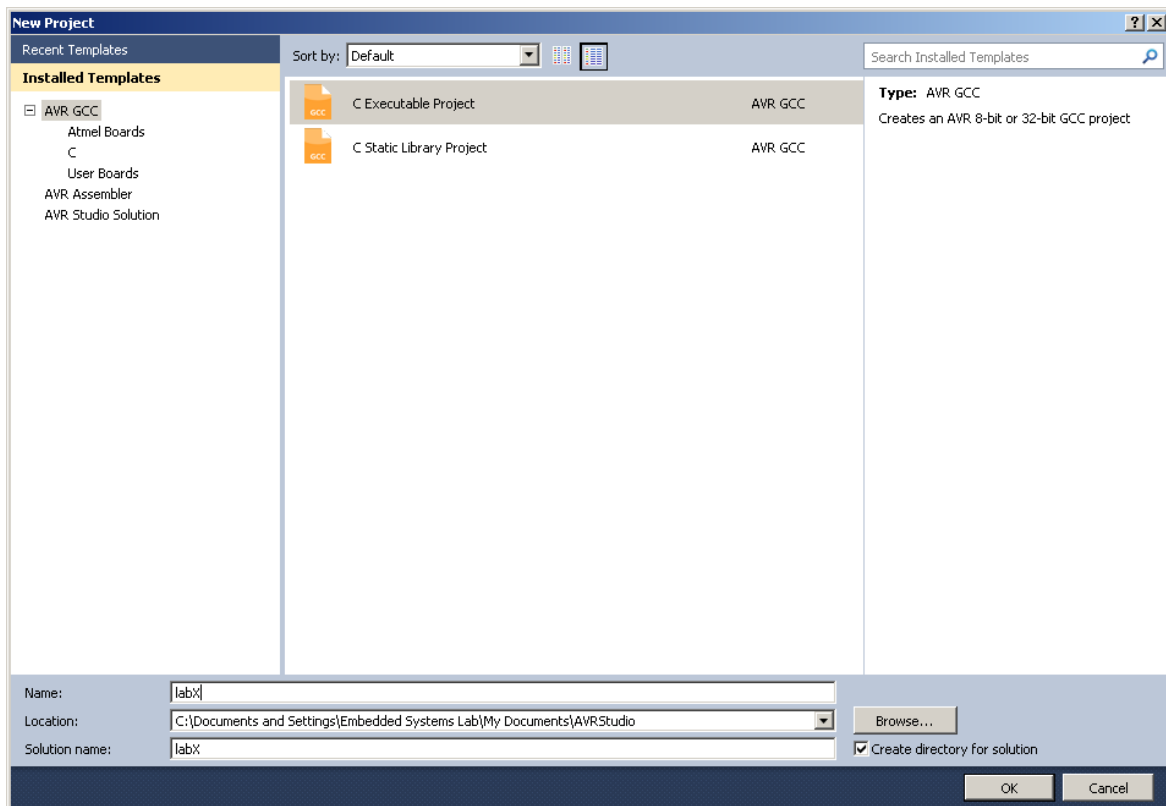
1. Open AVR Studio 6 from the Start Menu or the Desktop.
2. Select "Tools -> Customize -> Commands -> Keyboard -> Text Editor -> Plain Text -> Tabs" (from the menu at the top).
3. Select "Insert Spaces"

Creating, compiling, and simulating a new project

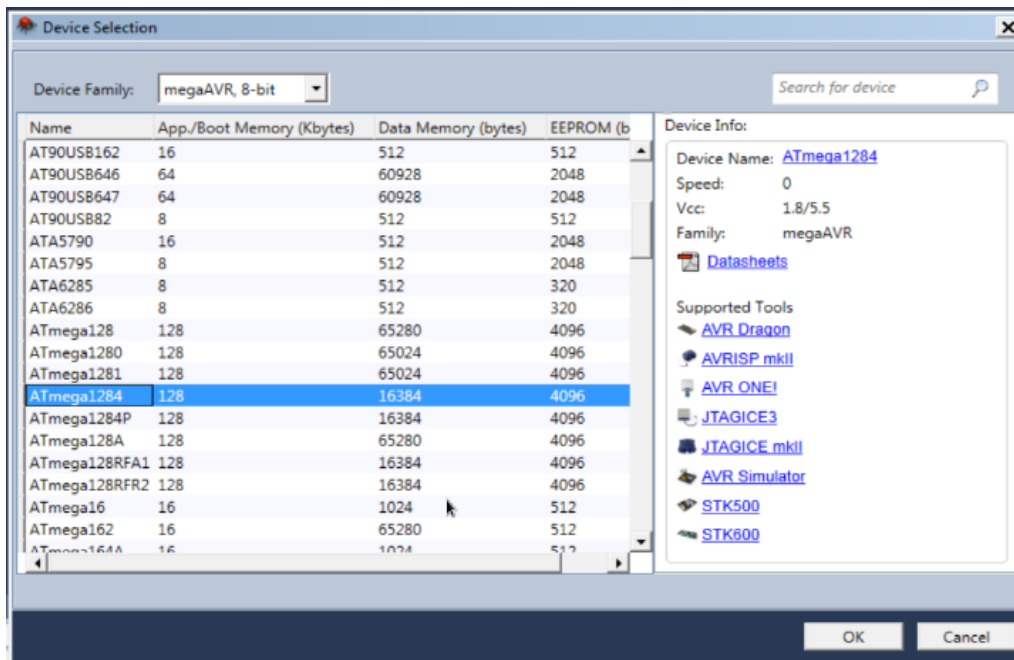
4. Open AVR Studio 6 from the Start Menu or the Desktop.
5. Select "File -> New -> Project" (from the menu at the top).

Note: Avoid the CS120B project template; it is outdated and designed for an old and outdated ATmega32 chip.

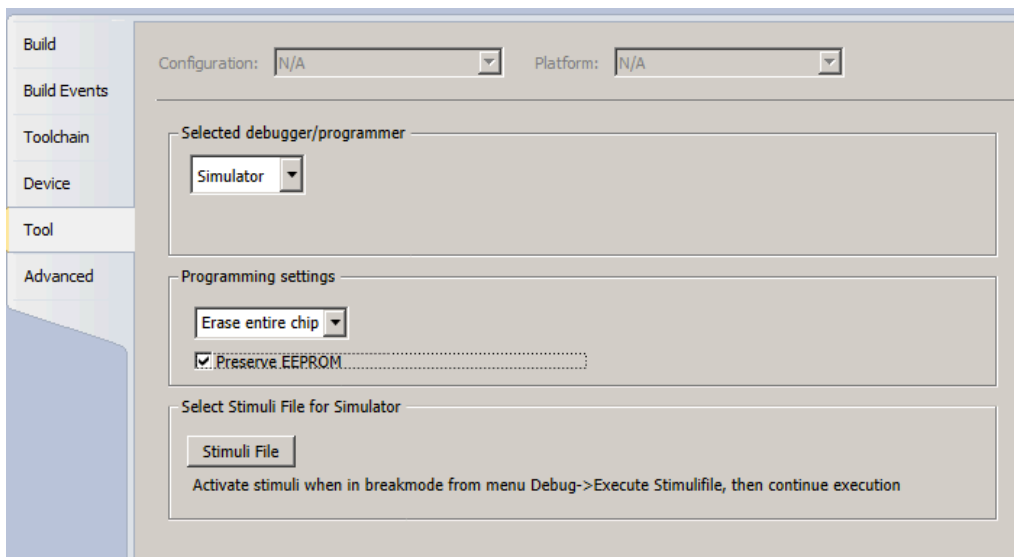
6. Select "C executable project" from the New Project window that pops up, type "[cslogin]_lab[#]_part[#]" in the name field near the bottom, filling in the appropriate values for this particular lab, then press "OK".



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 "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.) If the below window does not open, you can select "Project", then "Project Name" to open.



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:

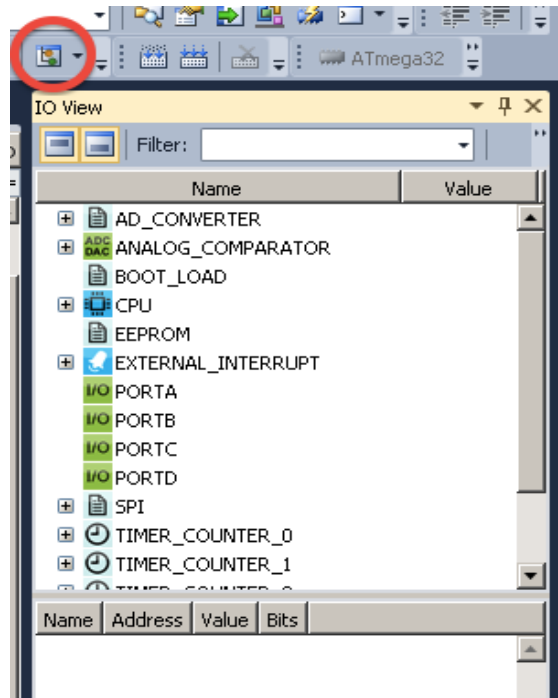
```
#include <avr/io.h>

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
    }
}
```

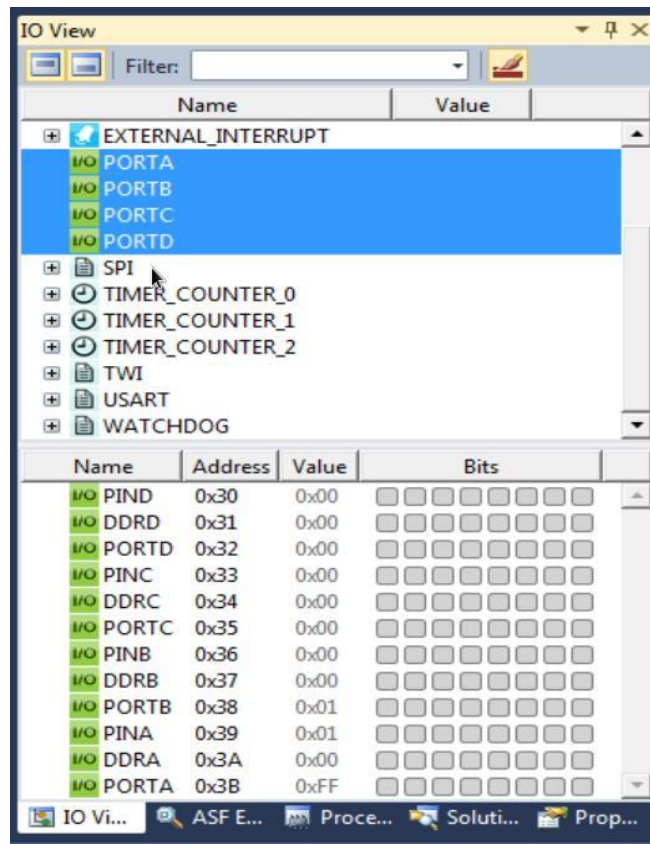
2. Select "Build -> Build solution". Check the Output window to ensure the build succeeded without any errors.

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

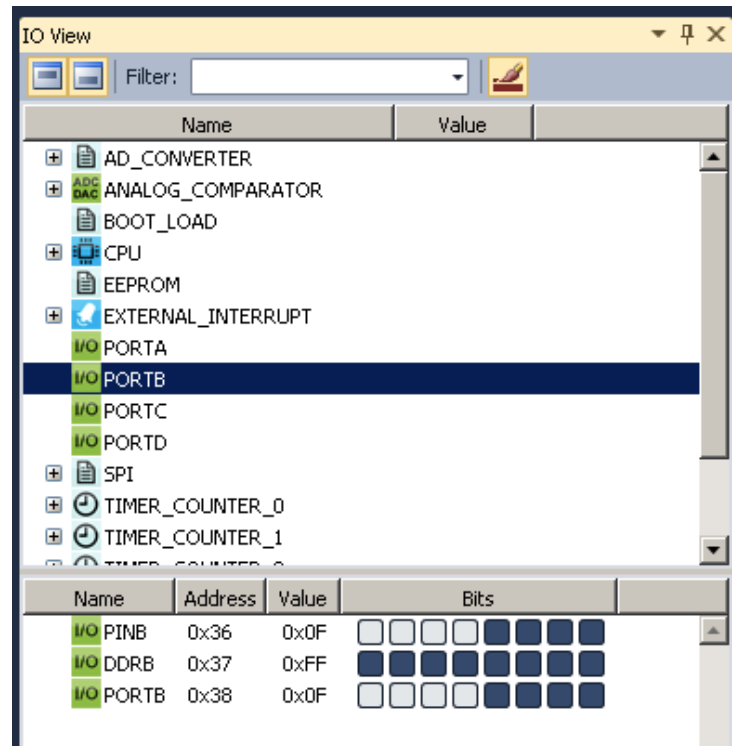
3. Select "Debug -> Continue".
4. Find and press the "I/O View" button (near the top, which has a symbol with a couple filled circles and a line), causing the "IO View" sub-window to appear.



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 | |

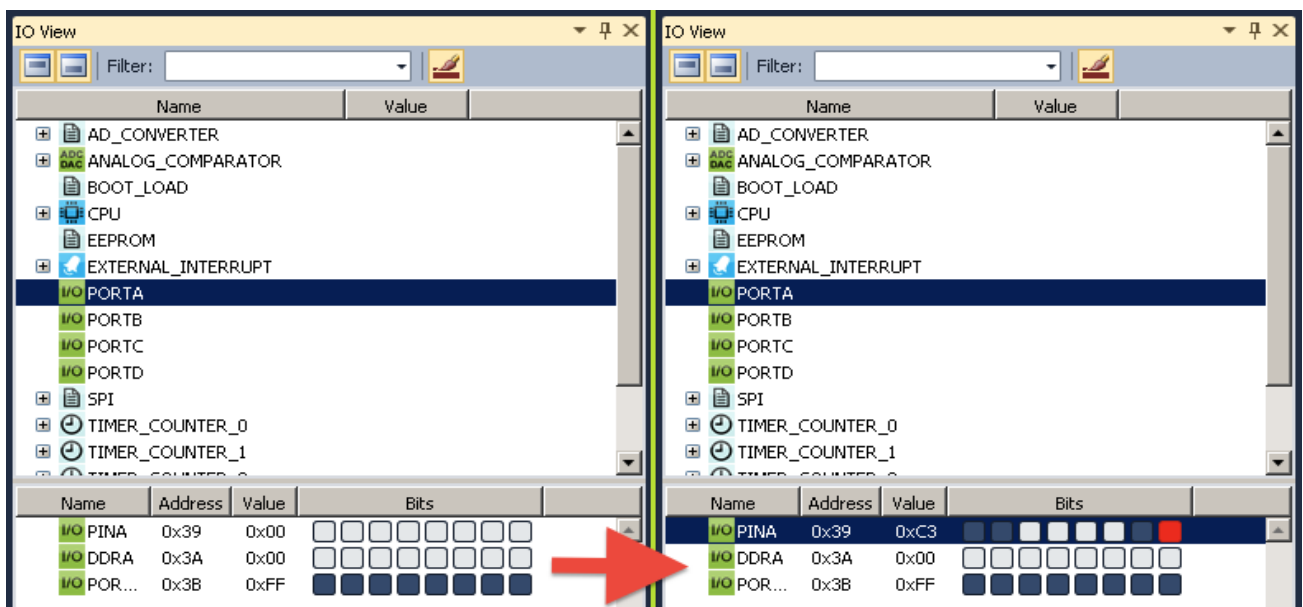
Reading from input pins

1. Modify the program into the following program:

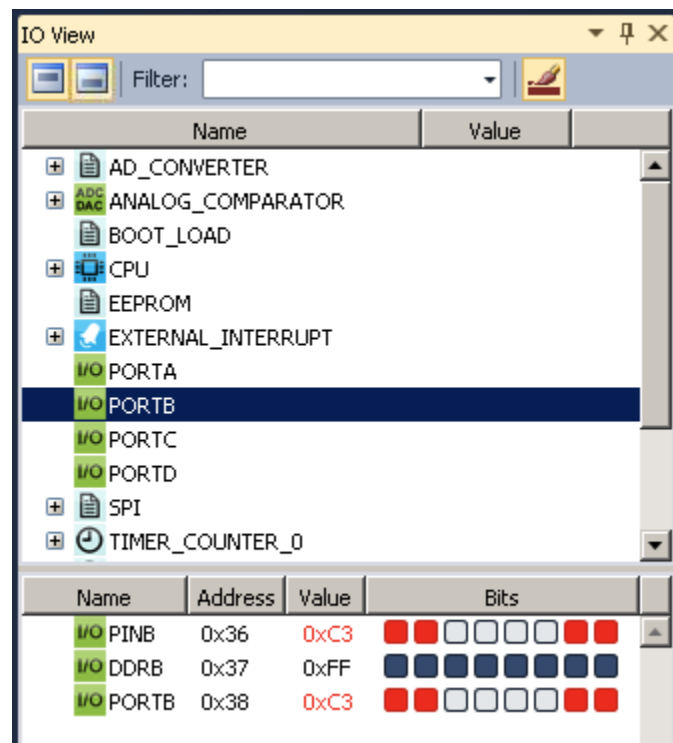
```
#include <avr/io.h>

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 now executes 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
        tmpA = PINA & 0x01;

        // 2) Perform computation
        // if PA0 is 1, set PB1PB0=01, else =10
        if (tmpA == 0x01) {

            // True if PA0 is 1
            // Sets tmpB to bbbbbb01
            // (clear rightmost 2 bits, then set to 01)
            tmpB = (tmpB & 0xFC) | 0x01;

        }
        else {

            // Sets tmpB to bbbbbb10
            // (clear rightmost 2 bits, then set to 10)
            tmpB = (tmpB & 0xFC) | 0x02;

        }
        // 3) Write output
        PORTB = tmpB;
    }
    return 0;
}
```

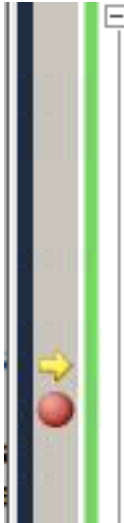
2. **NOTE:** Use of the notation "PA0", "PB1", and "PB0" in the comments to refer to port A's bit 0, port B's bit 1, and port B's bit 0, respectively. We will use such notation extensively in comments and lab assignment text, but realize that those are NOT recognized identifiers by the AVR C compiler.
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.



```
int main(void)
{
    DDRA = 0x00; PORTA = 0x00;
    DDRB = 0xFF; PORTB = 0x00;

    while(1)
    {
        // if PA0 is :
        if (PINA & 0x01)
            PORTB = (PORTB << 1) | 1;
    }
}
```

Bit Manipulation

Manipulating bits is an important aspect of programming embedded systems. This is quite different in comparison to programming an application for a desktop PC or a server. Additionally, making the code clear and readable is also very important. A good practice is to structure embedded applications as follows.

1. Read inputs and assign their values to variables
2. Perform the necessary computation on the input values
3. Write the result of the computation to the output port

The rationale for this structure is that many embedded applications exhibit time-oriented behavior. Reading all of the input bits at once ensures predictable behavior, as the input values may change if bits are read one-at-a-time. Similarly, it is preferable to write one value to all of the output bits, rather than writing them one-at-a-time, as doing so provides a stable output value. We will revisit these issues later in the quarter. Students are expected to adhere to this program organization throughout the duration of the quarter, both in lecture and in lab.

Consider the program P1 on the following page. Each iteration of the while(1) loop in the main function adheres to the structure outlined above.

This program makes extensive use of *bit manipulation* techniques, which makes extensive use of operators such as & (bitwise-AND), | (bitwise-OR), and hexadecimal constants (e.g., 0x01). If you are unfamiliar with any of these concepts, you should review Sections 2.4 – 2.7 of the Programming Embedded Systems (PES) zyBook; you may also wish to consult the “Programming in C” reference, which is available on iLearn under ‘Course Materials.’

The objective of bit manipulation is often to access individual bits in a larger word (e.g., an 8-bit unsigned char in ‘C’).

Program P2, shown two pages ahead, rewrites P1 to employ two useful functions: GetBit() and SetBit() which provides read/write access to the individual bits of an unsigned char. As long as the programmer is comfortable with the GetBit() and SetBit() functions, P2 is much easier to maintain.

In practice, GetBit() and SetBit() are useful when manipulating 1 or 2 bits. When manipulating more bits at a time it is easier, and cleaner, to use bit manipulation techniques such as *masking* and *shifting*, as discussed in Section 2.7 of the PES zyBook.

When reading the code, it is important to note that GetBit() and SetBit() make use of the ternary conditional operator (?:) in C. For details, please consult section 2.7 of the PES zyBook and/or the “Programming in C” reference.

// Program P1

#include <avr/io.h>

int main(void)

{

 // Configure port A's 8 pins as inputs, initialized to 1s

 // Configure port B's 8 pins as outputs, initialize to 0s

 DDRA = 0x00; PORTA = 0xFF;

 DDRB = 0xFF; PORTB = 0x00;

 // Intermediate variables used for port updates

 unsigned char tmpB = 0x00;

 unsigned char button = 0x00;

 while(1)

 {

 // 1. Read input

 button = PINA & 0x01;

 // 2. Perform Computation

 // if PA0 is 1, set PB1PB0=01, else =10

 if (button == 0x01) {

 // Sets tmpB to bbbbbb01

 // (clear rightmost 2 bits, then set to 01)

 tmpB = (tmpB & 0xFC) | 0x01;

 }

 else {

 // Sets tmpB to bbbbbb10

 // (clear rightmost 2 bits, then set to 10)

 tmpB = (tmpB & 0xFC) | 0x02;

 }

 // 3. Write the result to the output port

 PORTB = tmpB;

 }

}

// Program P2

#include <avr/io.h>

// Bit-access functions

```
unsigned char SetBit(unsigned char x, unsigned char k, unsigned char b) {  
    return (b ? x | (0x01 << k) : x & ~(0x01 << k));  
}  
unsigned char GetBit(unsigned char x, unsigned char k) {  
    return ((x & (0x01 << k)) != 0);  
}
```

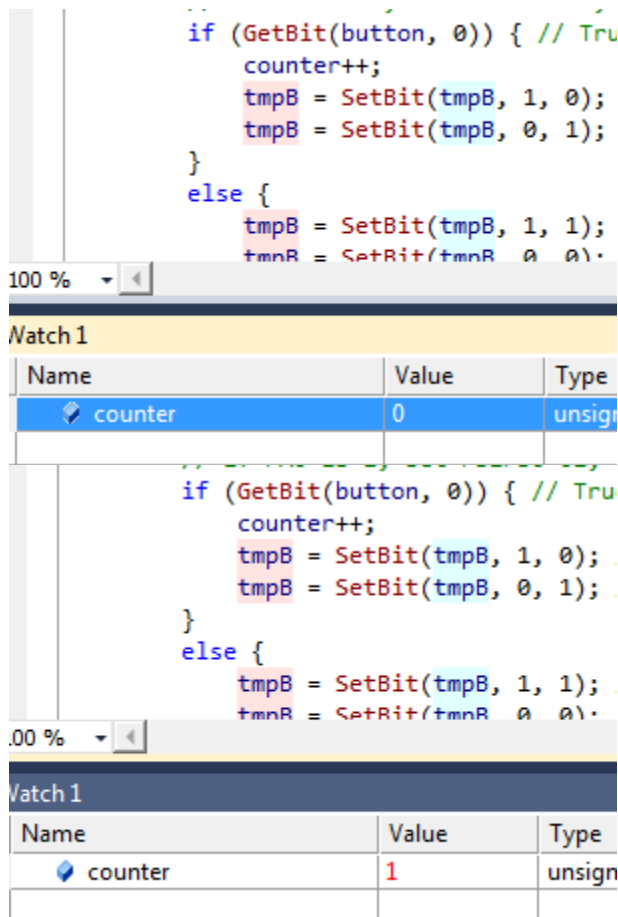
int main(void)

```
{  
    // Configure port A's 8 pins as inputs, initialized to 1s  
    // Configure port B's 8 pins as outputs, initialize to 0s  
    DDRA = 0x00; PORTA = 0xFF;  
    DDRB = 0xFF; PORTB = 0x00;  
  
    // Intermediate variables used for port updates  
    unsigned char tmpB = 0x00;  
    unsigned char button = 0x00;  
  
    while(1)  
    {  
        // 1. Read input  
        button = PINA & 0x01;  
  
        // 2. Perform Computation  
        // if PA0 is 1, set PB1PB0=01, else =10  
        if (GetBit(button, 0)) {  
            tmpB = SetBit(tmpB, 1, 0); // Set bit 1 to 0  
            tmpB = SetBit(tmpB, 0, 1); // Set bit 0 to 1  
        }  
        else {  
            tmpB = SetBit(tmpB, 1, 1); // Set bit 1 to 1  
            tmpB = SetBit(tmpB, 0, 0); // Set bit 0 to 0  
        }  
  
        // 3. Write the result to the output port  
        PORTB = tmpB;  
    }  
}
```

Note: A handy debug tactic is to use the watch list. Any variable you define can be added to the watch list. To do so, build your code, enter debug mode and 'break all'. You can then right click on your variables and add them to the watch list.

 Add Watch

As you step through the code you can then keep track of the value of your variables, such as 'counter' in the example below. Once you have stepped past the line of code and it has executed, any changes to your variables will be marked in red. Additionally you can manually edit the watch list values, which will change the current value of the variable within the program's execution. This is handy to test loop and branching conditions.



The screenshot shows a debugger interface with two panels. The top panel displays C++ code with a conditional statement. The bottom panel shows the 'Watch 1' list, which tracks the variable 'counter'.

```
if (GetBit(button, 0)) { // True
    counter++;
    tmpB = SetBit(tmpB, 1, 0);
    tmpB = SetBit(tmpB, 0, 1);
}
else {
    tmpB = SetBit(tmpB, 1, 1);
    tmpR = SetBit(tmpR, 0, 0);
}
```

100 %

| Name | Value | Type |
|---------|-------|--------------|
| counter | 0 | unsigned int |

0.00 %

| Name | Value | Type |
|---------|-------|--------------|
| counter | 1 | unsigned int |

Exercises

For each exercise, create a new project named [cslogin]_lab1_part1, [cslogin]_lab1_part2, etc. Each part should be demoed to a TA, including Challenge exercises (if attempted).

Completion of Exercises 1-6 is mandatory, and is worth 80% of the scores for the laboratory assignment. Completion of the challenge problems (below) is optional, and is worth the remaining 20%.

Write C programs for the following exercises using AVR Studio targeting an ATmega1284 microcontroller. Use the GetBit and/or SetBit functions when appropriate. These functions all employ combinational logic (e.g., inside a while(1) loop) and do not require state machines.

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. In this laboratory exercise, we will not actually use sensors or LEDs; just use the simulator and set inputs to 0 and 1 as needed and observe the output.

PA0 = garage door sensor (input)

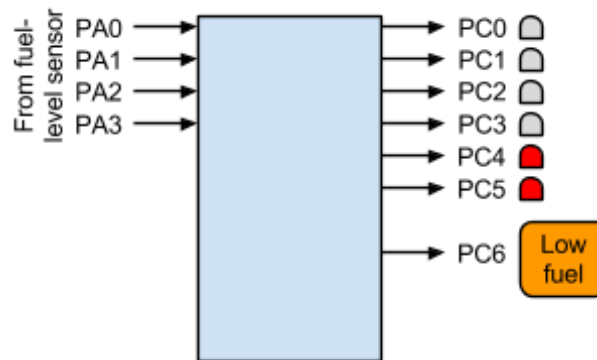
PA1 = light sensor (input)

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 program in Exercise 2 to still write the available spaces number, but only to PC3...PC0, **and** to set PC7 to 1 if the lot is full.
4. Count the number of 1s on ports A and B and output that number on port C.

5. 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. Level 5-6 lights PC5..PC3. 7-9 lights PC5..PC2. 10-12 lights PC5..PC1. 13-15 lights PC5..PC0. Also, PC6 connects to a "Low fuel" icon, which should light if the level is 4 or less. (The example below shows the display for a fuel level of 3). In this laboratory exercise, we will not actually use LEDs; just use the simulator and set inputs to 0 and 1 as needed and observe the output.



6. In addition to the above, PA4 is 1 if a key is in the ignition, PA5 is 1 if a driver is seated, and PA6 is 1 if the driver's seatbelt is fastened. PC7 should light a "Fasten seatbelt" icon if a key is in the ignition, the driver is seated, but the belt is not fastened.

Note: A port can be set to be input for some pins, and output for other pins by setting the DDR appropriately, e.g. `DDRA = 0xF0`; will set port A to output on the high nibble, and input on the low nibble.

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

Challenge Problems

7. 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.)

8. Read an 8-bit value on PA7..PA0 and write that value on PB3..PB0PC7..PC4. That is to say, take the upper nibble of PINA and map it to the lower nibble of PORTB, likewise take the lower nibble of PINA and map it to the upper nibble of PORTC (PA7 -> PB3, PA6 -> PB2, ... PA1 -> PC5, PA0 -> PC4).
9. A car's passenger-seat weight sensor outputs a 9-bit value (ranging from 0 to 511) and connects to input PD7..PD0PB0 on the microcontroller. If the weight is equal to or above 70 pounds, the airbag should be enabled by setting PB1 to 1. If the weight is above 5 but below 70, the airbag should be disabled and an "Airbag disabled" icon should light by setting PB2 to 1. (Neither B1 nor B2 should be set if the weight is 5 or less, as there is no passenger).