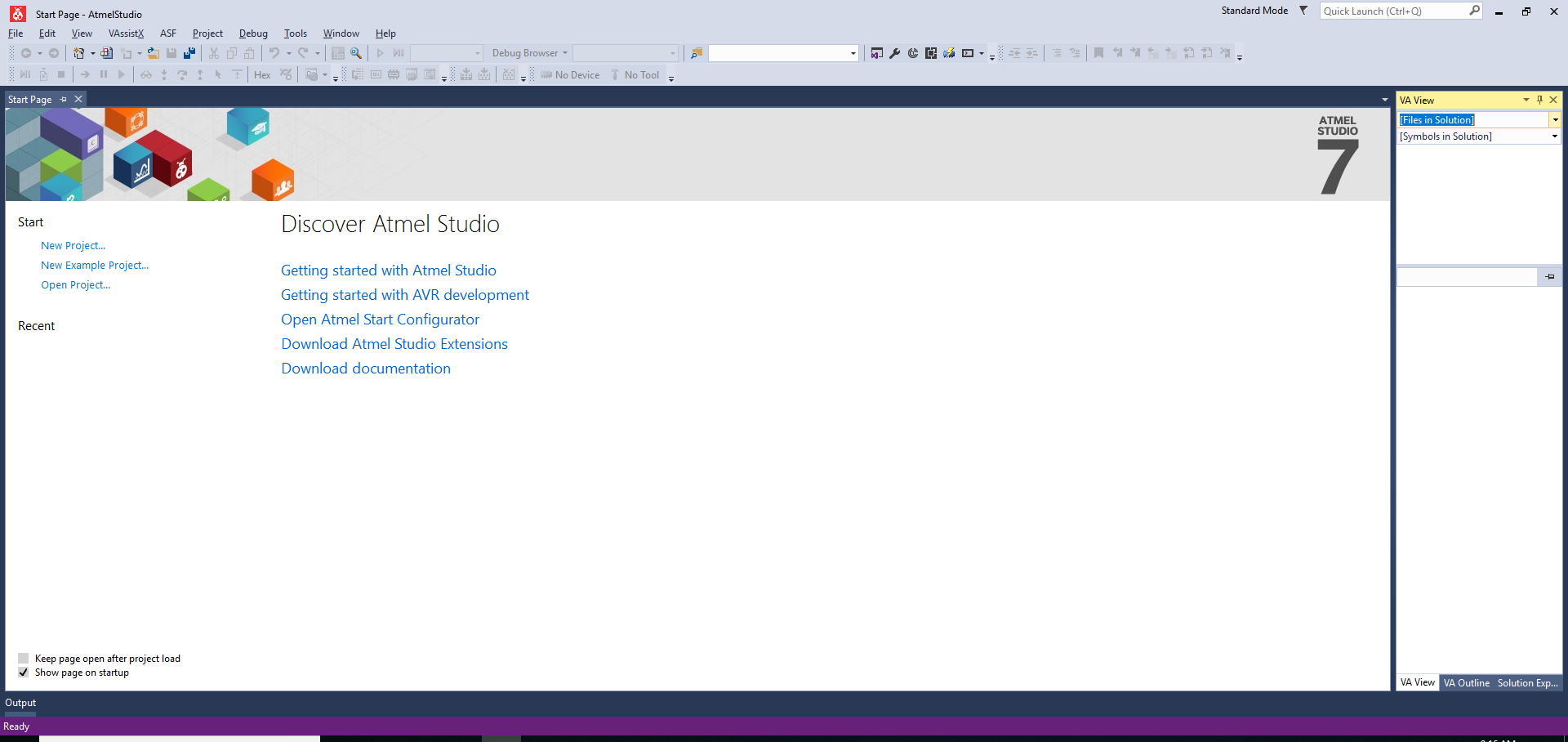
## Intro to Atmel Studio 7 software

**Important**: You are not able to run the tests in Atmel studio. If you are developing on Atmel studio, you will still need to run the tests on the lab machines for the first week.

If you are working on Windows, you can program AVR microcontrollers using a software called Atmel Studio (version 7 as of June 2019). This walkthrough introduces Atmel Studio 7, including the C editor, compiler, simulator, and debugger functionalities.



### Github Desktop

It is recommended that you save your work to a version control system (like Github). You can download the GitHub client “[Github Desktop](https://desktop.github.com/).” This provides an easy to use interface to upload your code to GitHub so you are able to save versions of it and work from both the lab machines and your desktop or laptop.

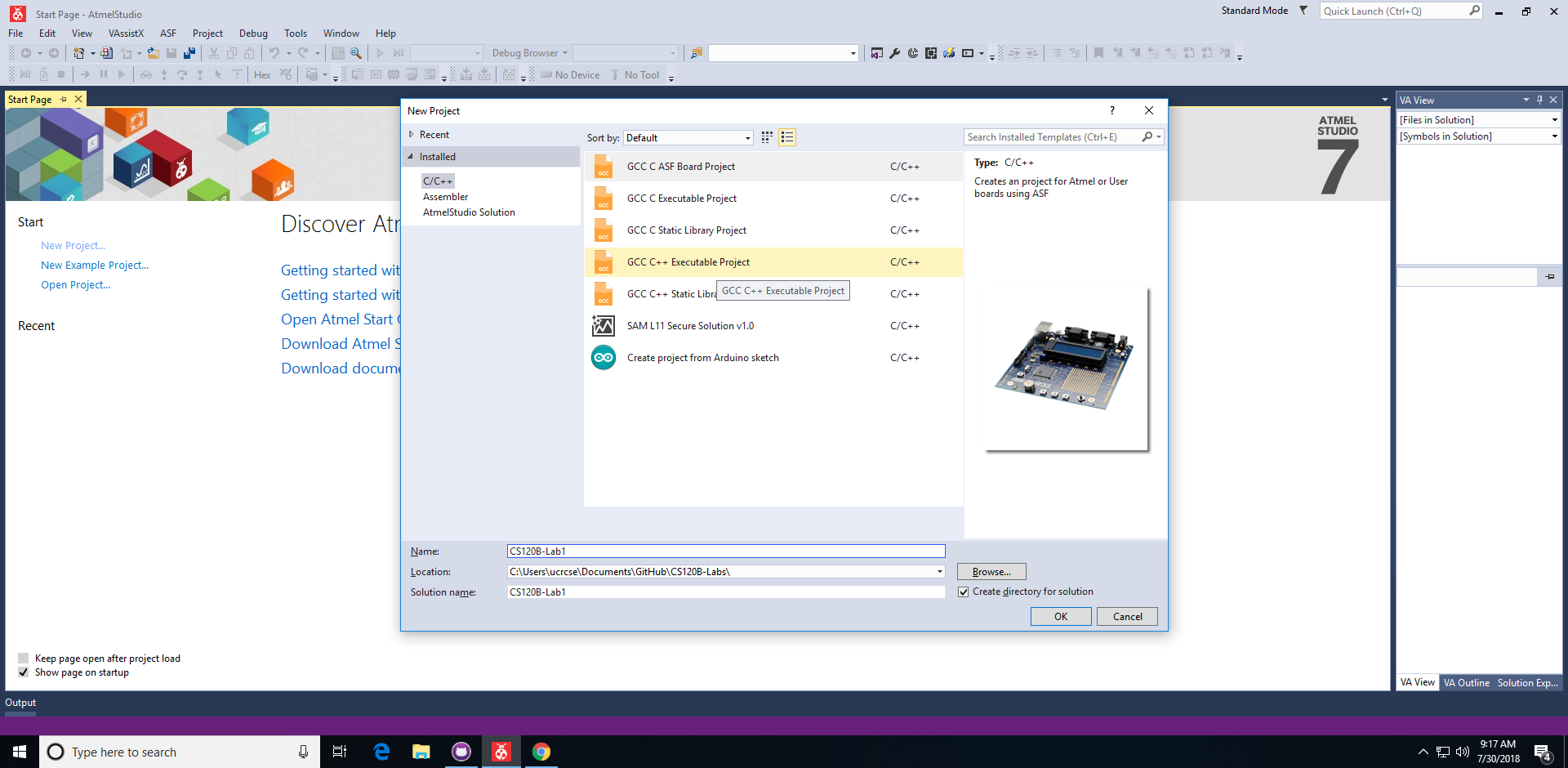
### Using spaces instead of tabs.

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.

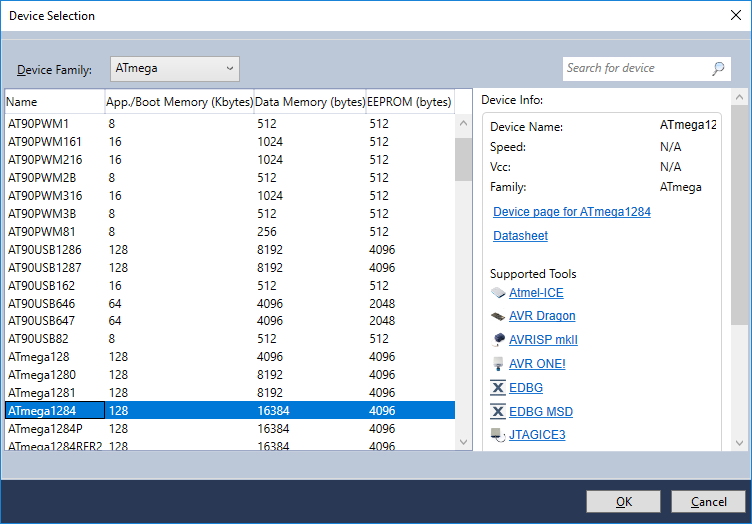
1. Open Atmel Studio 7 from the Start Menu or the Desktop.
2. Select "Tools -> Customize -> Commands -> Keyboard -> Text Editor -> Plain Text -> Tabs" (from the menu at the top).
3. Select “Insert Spaces”

### Creating, compiling, and simulating a new project

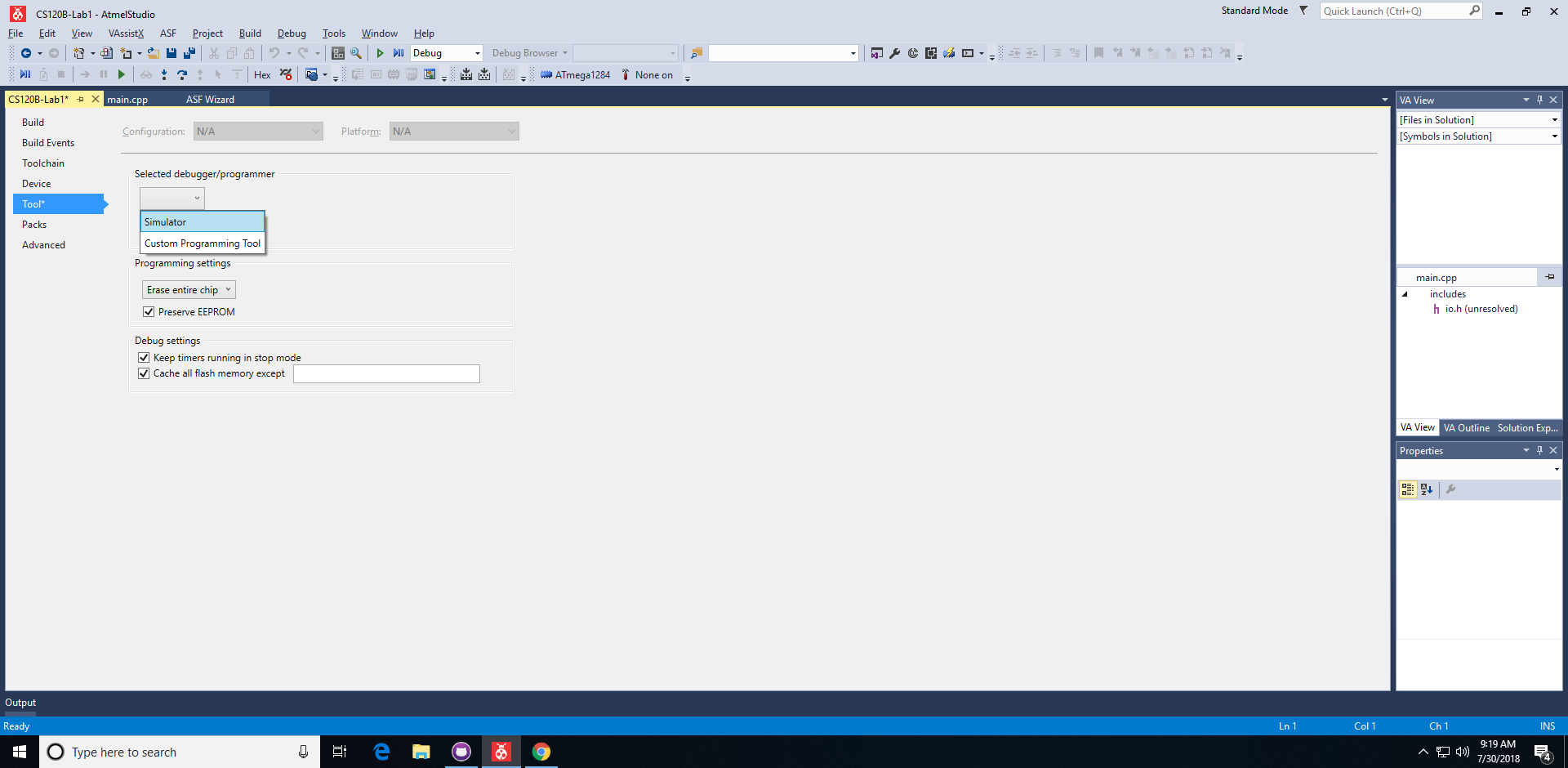
1. Open Atmel Studio 7 from the Start Menu or the Desktop.
2. Select "File -> New -> Project" (from the menu at the top).
3. Select "GCC C Executable Project" from the New Project window that pops up, type "[cslogin]\_lab[#]\_part[#]" in the name field near the bottom, filling in the appropriate values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



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



1. A sample program appears with an empty "while(1)" loop. Select "Build -> Build solution".
2. 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 behaviour 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.)



1. 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

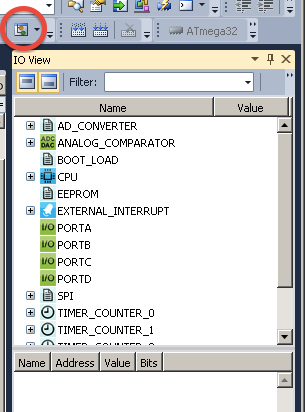
1. Replace the sample program by the following program (explained below) that sets port B's 8 pins to 00001111:

|  |
| --- |
|  |

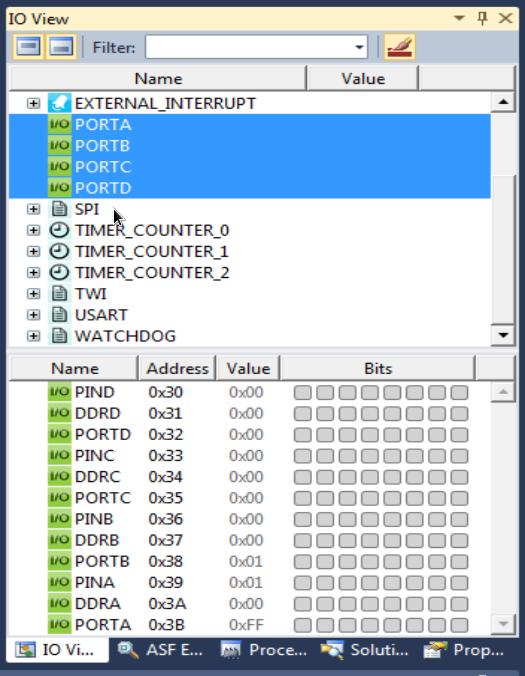
1. Select "Build -> Build solution". Check the Output window to ensure the build succeeded without any errors.

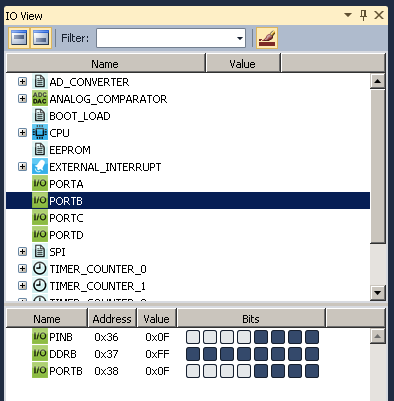


1. Select "Debug -> Continue".
2. Find and press the "I/O View" button (near the top, which has a symbol with a couple filled circles and a line), causing the "IO View" sub-window to appear.



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

1. 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.  
     
   
2. 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).



1. 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.

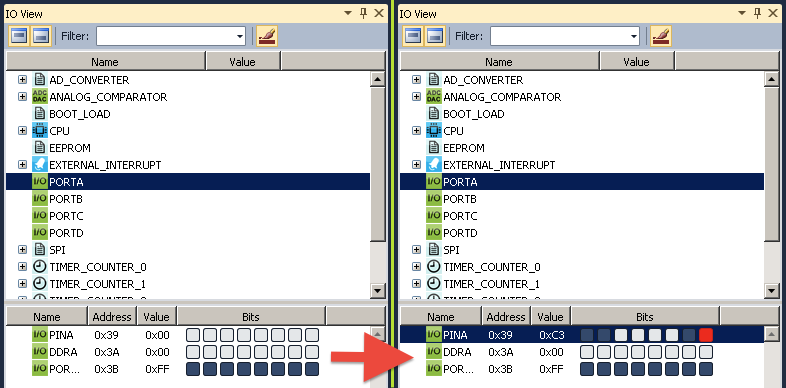
|  |  |  |
| --- | --- | --- |
| Icon | Name | Shortcut |
|  | Start Debugging and Break | Alt+F5 |
|  | Stop Debugging | Ctrl+Shift+F5 |
|  | Start Without Debugging |  |
|  | Continue | F5 |
|  | Restart | Ctrl+F5 |
|  | Break All |  |
|  | Step Into | F11 |
|  | Step Over | F10 |
|  | Reset | Shift+F5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

### Reading from input pins

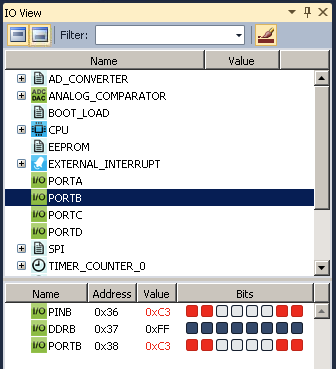
1. Modify the program into the following program:

|  |
| --- |
|  |

1. Select "Build -> Build solution". Ensure the build succeeded without any errors.
2. Select "Debug -> Continue".
3. Select "Debug -> Break all".
4. 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.



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



1. You can change PINA again, continue, break all, and view port B again and note that B should match A.
2. 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:

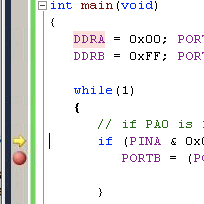
|  |
| --- |
|  |

1. **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.
2. **NOTE:** Use of a temporary variable tmpB instead of reading from PORTB.
3. 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 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.



### 

### 