

AVR Studio Simulator Introduction and Exercises

LEARNING OBJECTIVES

- get a brief introduction to most of the major debugging features AVR Studio
- read through a refresher on C programming and an introduction to I/O on AVR microcontrollers
- gain an appreciation of what the AVR Studio simulator is and what it's good for
- learn how to use the AVR Studio user interface to control program execution and to simulate I/O

COMPONENTS REQUIRED

- a Windows PC (or a Mac running VMware Fusion, or Wine on Linux)
- downloaded AVR Studio installation package, "AVR Studio 4.18 (build 684)" from http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
- downloaded AVR Studio service pack, "AVR Studio 4.18 SP2", also from http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
- downloaded AVR C compiler toolchain, <http://sourceforge.net/projects/winavr/files/WinAVR/20100110/WinAVR-20100110-install.exe/download>

PREPARATION

For both a C-language refresher as well as an introduction to the basics of programming for the Atmel AVR ATmega microcontrollers, it is highly recommended that you take the time to go through the “C Programming” chapter of the tutorial at <http://www2.tech.purdue.edu/ecet/courses/referencematerial/atmel/>. This document provides a very, very good foundation for almost all of the microcontroller-related material for the semester, so now's the time to start dealing with it. You can probably skip the final subsection on Pointers if you care to, as you won't really be needing them this semester. The other chapter, “References”, should be near the top of your list of resources around about Week 5 of this course.

One issue with the aforementioned Purdue tutorial is that at the time of writing, they were using the CodeVisionAVR C compiler, which we are not using. Features specific to that compiler, in particular the “Single Bit Manipulations” described in the “I/O Operation” section, and the “RAM vs. Flash” in the “Pointers” section, are not available in the compiler that we are using. However both can be done, but use a different mechanism in WinAVR.

INTRODUCTION

It is often believed that without target hardware it is difficult, if not impossible, to develop and test software for a microcontroller (a.k.a. *MCU*) project. This is often not the case as many of the microcontroller manufacturers (or their partners) offer software simulators that exist for just this purpose. Not only does Atmel's free IDE (Integrated Development Environment), AVR Studio, provide the framework for compiling programs and downloading them to the MCU, but

it also comes with the ability to simulate programs for most of their AVR microcontrollers. This simulator has the ability to not only execute AVR instructions but also to simulate limited digital I/O (input/output). So it's not the case that if you don't have an STK500 (the hardware development board that you'll be using this semester) in front of you that you can't be writing and testing software for homework, labs or final projects.

AVR Studio should feel relatively similar to other IDEs that you may have used in your introduction to programming classes, like Microsoft Studio Express C, Eclipse, or NetBeans. IDEs all look pretty much the same at 20,000 feet. They all give you an easy way to edit files, to describe and build projects, and to execute and (usually) debug them. In most respects, once you've seen one IDE you've seen them all. The trick is knowing where all of the options are hiding for the things that you need to set, how clearly they're organized, and how well they perform for what's most important to you. One of the best things that AVR Studio (and WinAVR, which actually provides the Windows version of the compiler tools) has going for it is that it's free and that it performs pretty well – it's reasonably fast and doesn't seem to crash very often.

Although you won't be using one this semester (unless you spring \$100-250 for one yourself) AVR Studio can also be used with a USB device called an AVR JTAG programmer/debugger to debug real running hardware from your PC using exactly the same interface as this simulator. This AVR JTAG device is an interface between AVR Studio, your PC's USB port and a 6-pin (or 10-pin) connector on your target hardware board. It allows you to view and change any piece of data inside the running MCU, as well as to control program execution in exactly the same way that you'll be doing in the AVR Simulator. So by learning how to use the AVR Simulator, you'll also be learning how in-circuit debugging works on real hardware should you ever do that in the future.

The way that AVR Studio (and most IDEs) is organized is that one works on *projects*. A project is a collection of files, their interrelationships, and configuration settings necessary to build source, header files and libraries together into a final unit. This final unit is usually a program to be simulated or downloaded to a chip for execution. A project is composed of a list of files to be compiled, the compiler settings, the configuration of the chip being developed for, as well as the layout and contents of the various windows on the screen, etc. Just as changes to files need to be saved to disk, any changes to a project also need to be saved, otherwise all of this (meta)information is lost and will have to be entered again. So don't forget to save project settings as well as your source files (assuming you don't really want to discard project settings) when exiting AVR Studio.

On shared lab computers it is additionally helpful to give your project a more unique name than "Lab2", like "Lab2Fri", so that you can more easily identify it. That way you can differentiate your project from others' "Lab2". There's nothing funnier than watching a lab group make change after change to their code, recompile and re-run it over-and-over again, and always have it behave exactly the same way. It's because they keep downloading someone else's old lab code with the same name to their chip. Priceless! :-)

Don't forget to copy your entire project directory off to a flash stick or email them off the machine when you're done. It's not really recommended to work right on the flash drive because they're considerably slower than the internal hard disk. The lab machines do get periodically wiped-clean, so what's there one week may not be there the next.

About the simulator in particular, it should be fairly obvious that it's not possible to directly hook-up hardware like switches, LEDs, or motors and the like as there's no physical hook-ups to the simulator. However, one is able to look at a representation of the digital portion of the (virtual) physical interface to the chip with the simulator. For example, you can look at a physical output line on the controller to see that an LED would/wouldn't be lit if one were connected. Similarly for input, one can direct the simulator to behave as though a given digital pin were high/low as if a switch were currently open/closed.

Microcontroller programs produce outputs dependent on the program's current state and inputs. A simulated stream of inputs can be provided for every general-purpose digital input pin on the MCU being simulated. This is done by providing what is called a stimulus file, which is really just a time-line containing the state of some or all of the inputs at every point in time. This feature will be presented in a later section.

PROCEDURES

Installing and Running

First, download and install the software (as *administrator*) in the order listed in the **COMPONENTS** section above onto your PC. Reboot the PC afterward if you're prompted to do so for the installation to complete properly.

Once you have the software loaded on your machine, you should be able to go ahead and run AVR Studio. If it didn't create a desktop icon when it was installed, you'll just have to rough it and find it in the *Windows Start* menu.

Creating a Project

Once AVR Studio is running you should be prompted for a project to work with, either by creating a new one, or by using an existing project. In this case, click **New Project**.

Select *Project type* **AVR GCC**, enter *Project name* **simcounter**, select both *Create initial file* and *Create folder* and select (and remember the location of !) a directory in which to put the new **simcounter** project directory, then click **Next**.

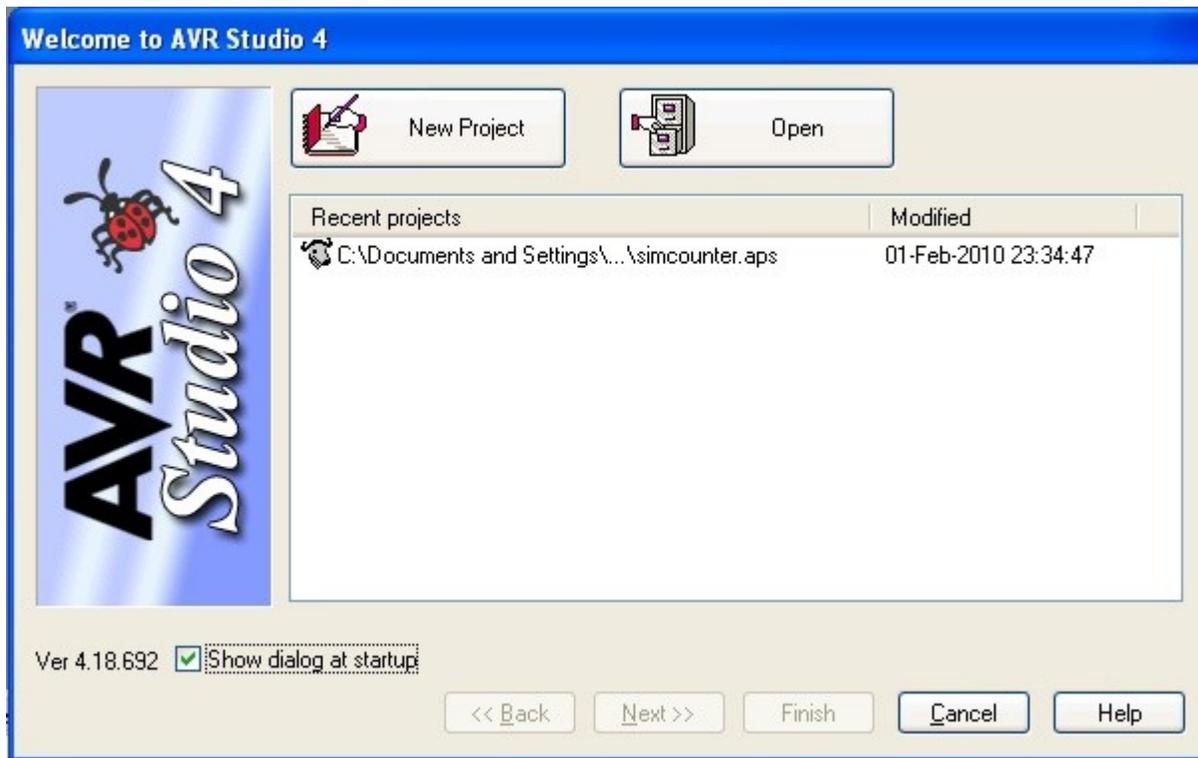


Illustration 1: Creating or opening existing project

Select *Project type* **AVR GCC**, enter *Project name* **simcounter**, select both *Create initial file* and *Create folder* and select (and remember the location of !) a directory in which to put the new **simcounter** project directory, then click **Next**.

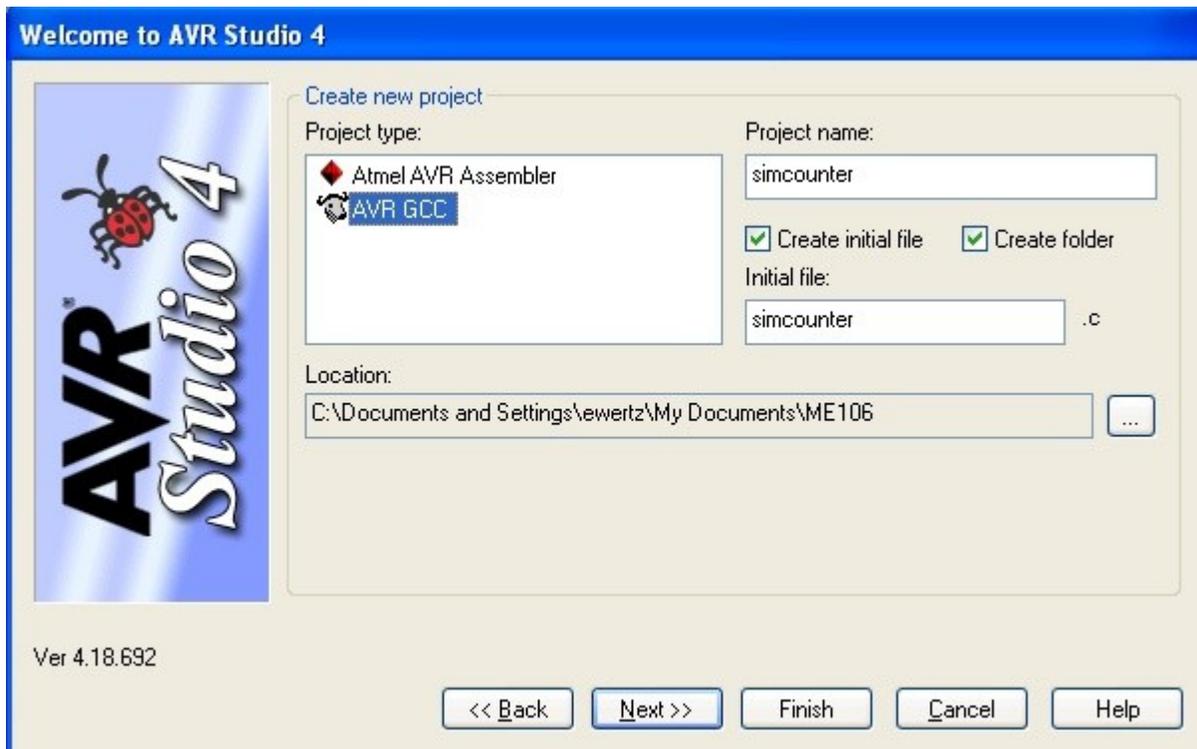


Illustration 2: Selecting C for the new project source type

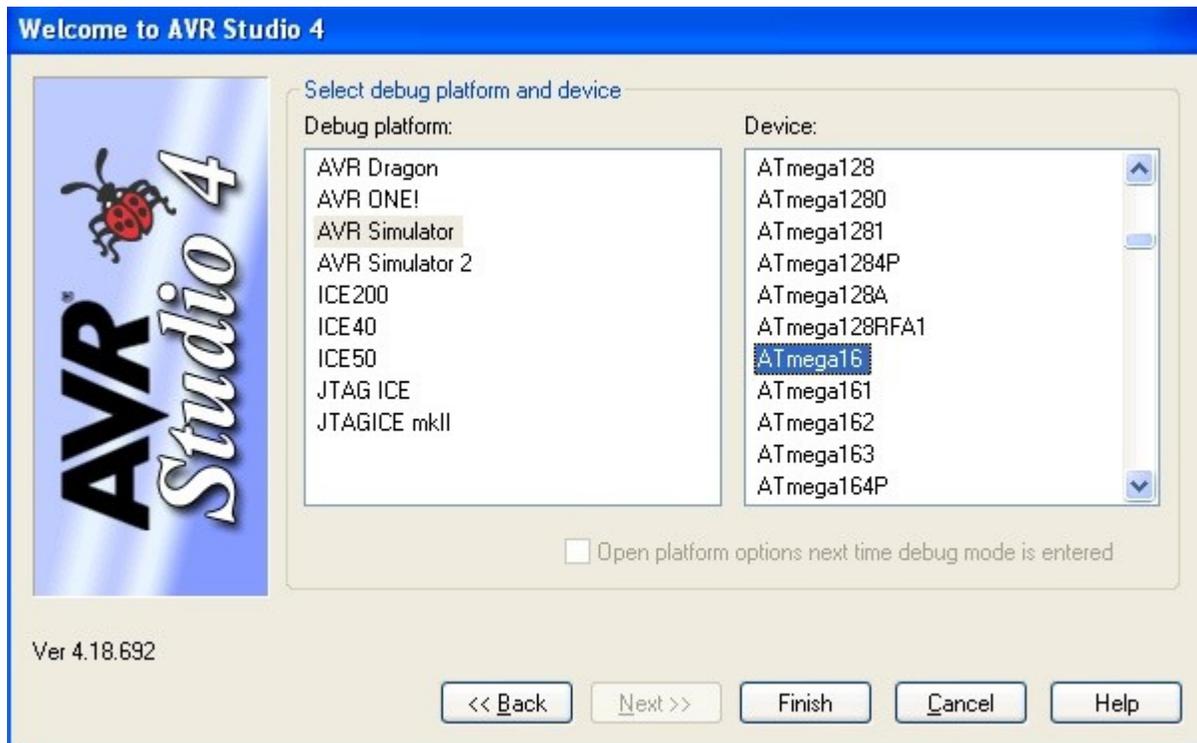


Illustration 3: Selecting the Simulator as debugger type for ATmega16

Select **Debug Platform AVR Simulator**, and **Device ATmega16**, then click **Finish**.

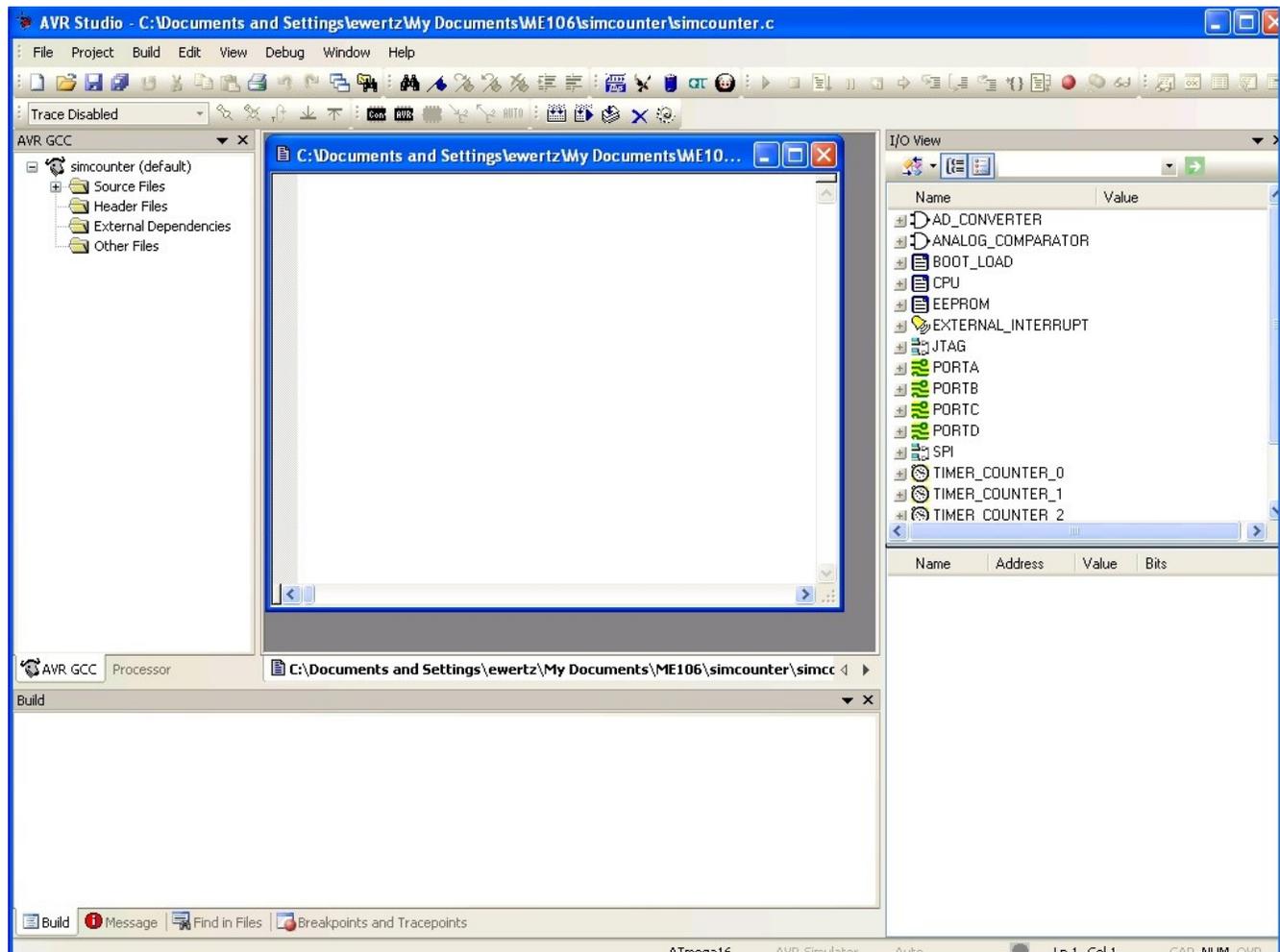


Illustration 4: Initial workspace layout with empty source file template

You should now see the default window layout in AVR Studio. Because you checked *Create initial file* when creating the project, AVR Studio has created a blank **simcounter.c** file and pre-opened this file for you. If you click on the expand-group icon (the Plus icon) in front of *Source Files* in the left-hand window, you'll find that **simcounter.c** has been added to the **simcounter** project. If you needed to include any additional files source files for your project, you would right-click on *Source Files* and choose the appropriate method to do this. You won't need to for this example, but you will for labs later in the semester.

At this point, cut-and-paste the contents of **simcounter.c** from the final section of this document into the blank window created for your project's **simcounter.c**. While you're at it, save the contents of this file by doing a **File->Save** (or typing **CTRL-S**, or pressing the floppy disk icon). Alternatively, if you already had a copy of **simcounter.c** on your computer, you could also have used *Add Existing File* and pointed it to the file's location to add it to the list of the project's source files. This is commonly done when you are given pre-written helper code to perform ancillary functions within the MCU, such as communicating with the UART (the serial line that connects back to the PC). This is one of the ways to include code from elsewhere to extend the function of your program.

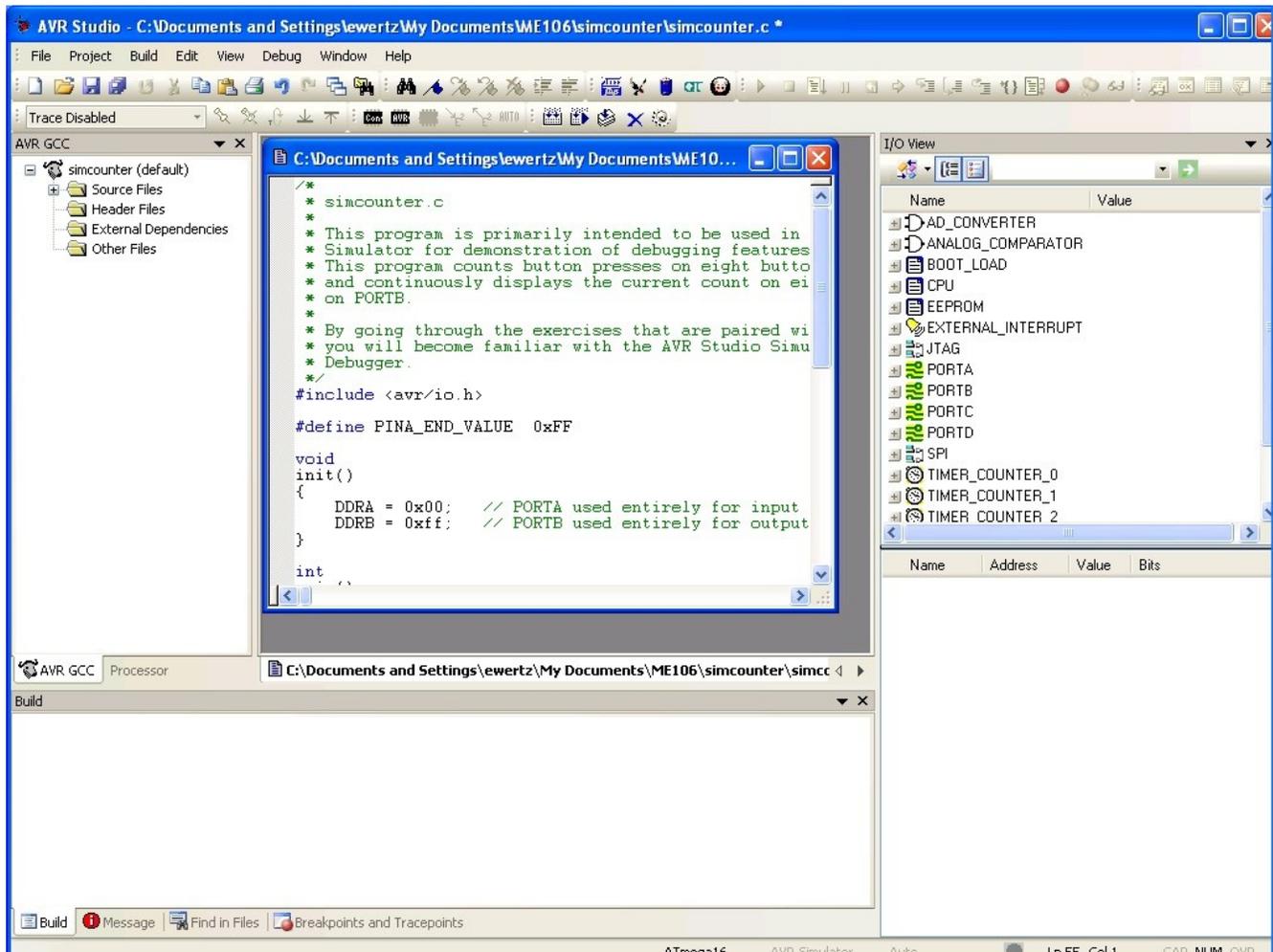


Illustration 5: Entering source code into project

Now you will add an additional data file to be used later when simulating I/O using this project's code. Create a new file called **simcounter-PORTA.sti** by right-clicking on *Other Files* in the left-hand window and select *Create New File*. Cut-and-paste the contents of the section titled **simcounter-PORTA.sti** at the end of this document into the window that was opened for this file, and similarly save it to disk.

Configuring and Building Projects

The process of *building* code for a microcontroller (and for a PC program, for that matter) is a many-part process of turning your source code and other provided code into a final executable program. The tool that translates your C code into microcontroller instructions is the *compiler*. The compiler takes a source file, typically a `.c` file, as input and generates an *object* file, typically a `.obj` or `.o` file. So, for example, a successfully compiled source file named `myfile.c` would generate the object file `myfile.obj`.

Another tool called the *linker*, takes all of your `.obj` files, and other `.obj` and `.lib` files that you may have specified, and combines them into the final program. On Windows, this final product is typically a `.exe` (executable) file, but for most microcontrollers it is a `.hex` file. It is this `.hex` file that is stuffed into the microcontroller over some type of serial/USB connection and finally executed after the microcontroller is reset. When all is said and done, making a hex file from one source file generates a number of auxiliary files along the way – that you don't necessarily know about.

AVR Studio is smart enough to know that when you change any individual source file, that it needs to use the compiler to generate a new `.obj` file from that `.c` file. It does this solely by noting that the `.c` file is more recent than its corresponding `.obj` file, and therefore needs to compile it. AVR Studio does not blindly (re)compile every source file every time a build is requested because that can substantially increase the time that it takes to generate the final `.hex` file, if for instance, your project was composed of many source files.

The problem is that AVR Studio can *not* be counted on to know when changes other than source file changes have been made that affect how object files or `.hex` files are built. These changes are typically settings changes, like the optimization settings, which microcontroller you're using (atmega16), at what frequency it is running, or which options are checked/unchecked, as shown in the screenshot below which we'll be discussing next. Even less obvious is the case where you take an older source file (like the last copy that you had working an hour ago before you made your more recent changes) and substitute it for your current one. Because this older file has a file timestamp that is an hour old, and the most recent corresponding `.obj` file was generated just minutes ago, AVR Studio can't know that you switched in a “newer older” source file and that it really does need to recompile it again. You can *Build* all day, but the compiler will not remake the object file for the source file that's there, and you'll never get to run the code that you're expecting to be there.

There are two ways to get around these types of build-synchronization problems, instead of just doing a regular *Build* (F7). You can either force everything to be rebuilt from scratch, or you can clean out all of these intermediate/temporary files and build them again. These two processes are *Rebuild All* and *Clean* respectively, and can be found under the *Build* menu. In general, I'd say that *Clean* is only useful when you're at the end of your lab and want to collect all of your work for emailing or thumb-driving because it gets rid of all of the intermediate by-product files that you don't care about (because the compiler will regenerate them) saving space and clutter.

That leaves *Rebuild All*. Do a *Rebuild All* when there's any doubt about what code it is that you want to be executing. There's no harm in doing a *Rebuild All* every time rather than just a *Build*, it's just that it'll take longer to generate the hex file. Just *always, always, always* make sure that code compiles without errors (otherwise you'll just download over-and-over the last hex file you successfully made) and downloads without incident to the microcontroller.

Be forewarned – chunks of time even larger than the chunks of hair you will be pulling out of your head can be saved merely by checking for reported errors (and copious use of Rebuild All) every step along the way.

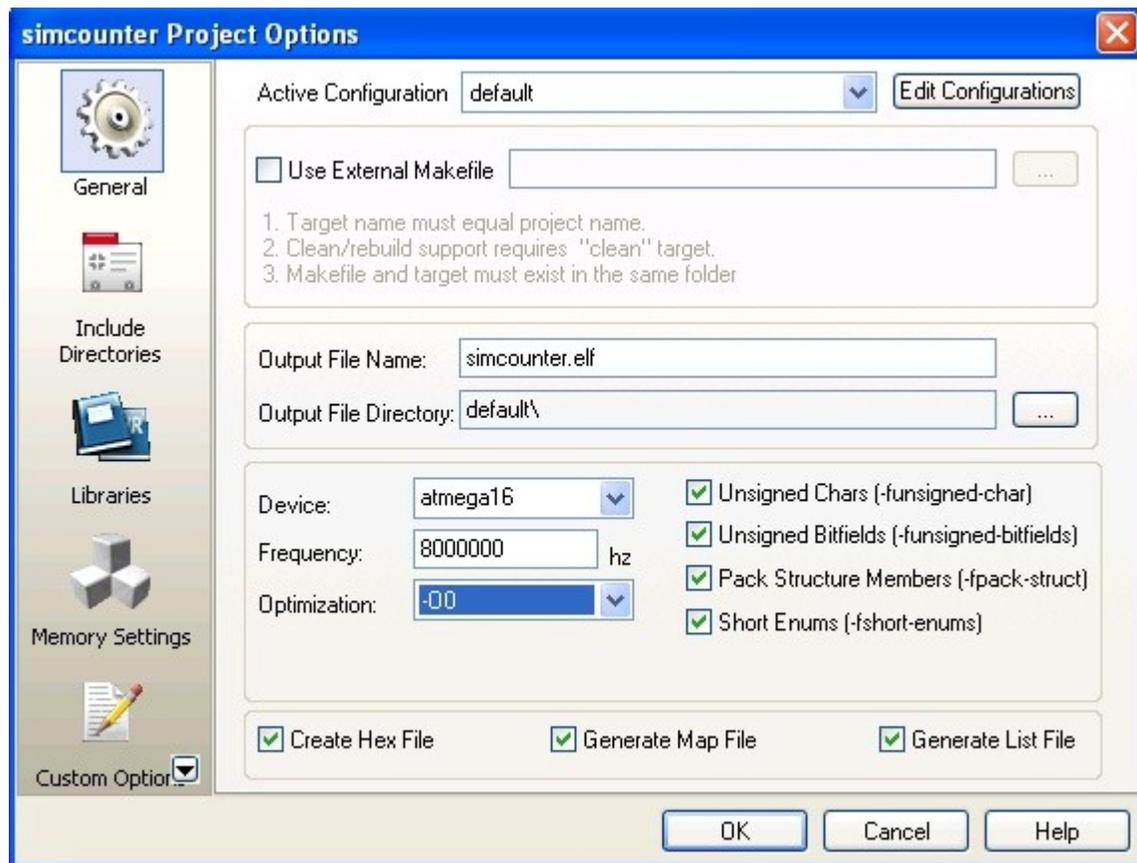


Illustration 6: Configuring project for build and MCU options

In *Project->Configuration Options->General*, set *Frequency* to **8000000** and *Optimization* to **-O0** (note that this is capital-oh zero) and, of course, click **OK**.

It should be noted that this optimization setting is usually *not* the most popular for generating code to be downloaded to a real MCU, but it is probably the best choice for generating code to be simulated. However, in subsequent weeks in the lab, once you're ready to start programming your microcontrollers in the lab, you'll often be directed to change this setting **-O2**. Just remember that should you need to revert to simulating code again, don't forget to change this option back to **-O0**, and then back again to **-O2** when building your code for the real MCU. If you ever start seeing strange behavior when stepping execution through simulated programs, it's probable that you weren't using **-O0** for the code to be simulated. What usually happens is that while stepping through your program, the cursor doesn't accurately advance from line to line as you'd expect it to. Please see the **Additional Information** section at the end of this document for more on code optimization.

At this point do a *Build->Rebuild All*, and you should be all set. Once again, doing a *Rebuild All* rather than just a *Build* is *always* a good idea after you have changed any project configuration options..

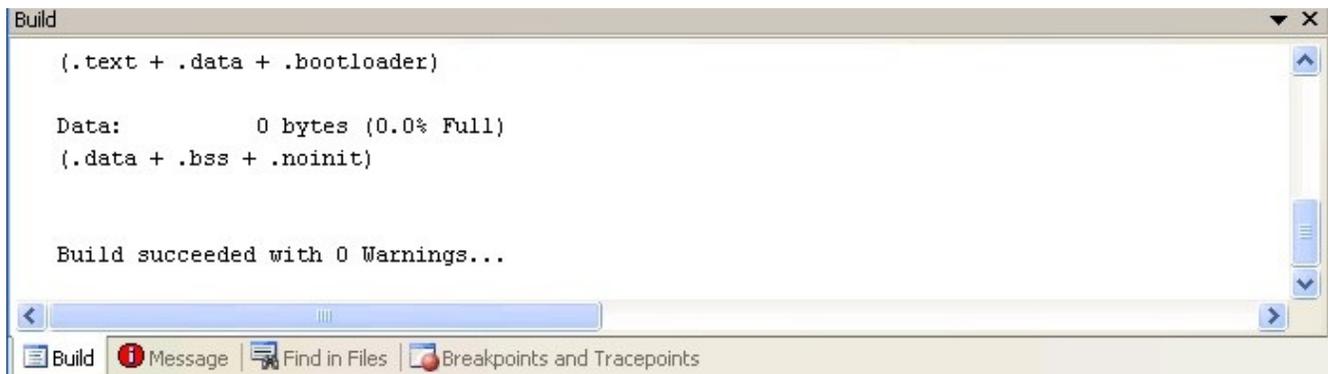


Illustration 7: Result of first build with cut-and-paste source code

When the build is complete, confirm that the last line in the Build window at the bottom of the screen is **Build succeeded with 0 Warnings...** as shown above. If this isn't what you see, somehow you boffed your cut-and-paste, so go back and try it again.

For some reason, there are certain configuration settings in AVR Studio that cannot be set unless you've already started simulating a program, and the following is one of them. To do so, initiate the simulation with *Debug->Start Debugging*. Now in *Debug->AVR Simulator Options* you should set *Frequency* to **8.00MHz**. Go figure..

Starting and Stepping through Execution

If you hadn't just done so immediately above, normally you'd start simulating the execution of the program with *Debug->Start Debugging*, or by pressing the green right-facing triangle



*Illustration 8:
Mr. Start
Debugging icon*

in the toolbar. The first thing that the simulator then does is to start-up and then stop before the very first simulated instruction is executed. It stops here to give you the chance to do all the setup you need before starting the run.

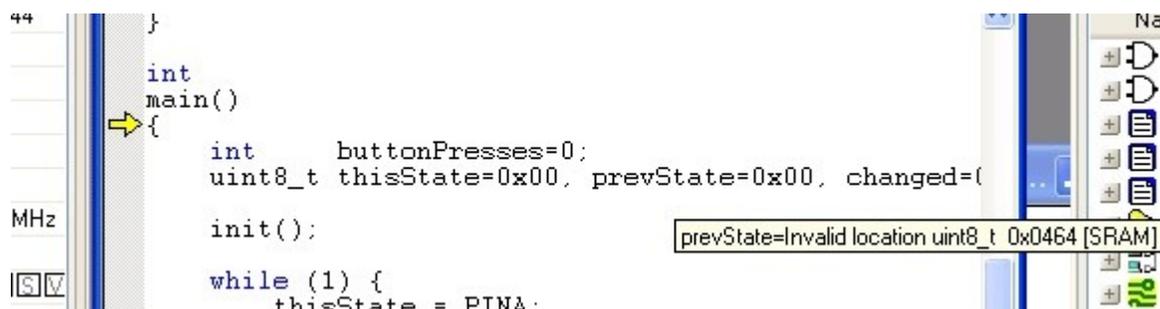


Illustration 9: Breakpoint at program start position

The important thing to notice at this point is that there should be a little yellow right-facing arrow pointing to the opening curly-brace of the function `main()`. This arrow indicates the *next* statement to be executed.

At any point when you have control of the running program (that is, when it isn't free-running at full-speed uninterrupted), you can step through the program slowly, line-by-line, at your own pace. There are three ways to step – *step into*, *step over*, and *step out*. Each of these three stepping actions has an icon in the toolbar to the right of the yellow arrow icon, shown below.

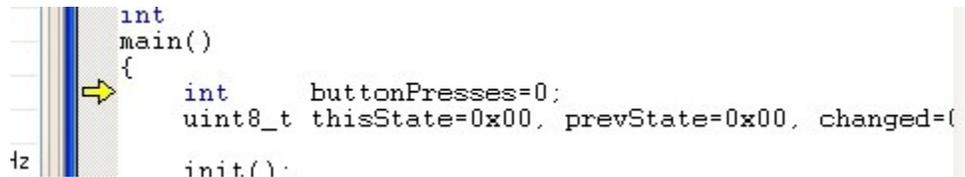


*Illustration
10: The three
Step sisters*

Step into and *step over* behave similarly except when a function call is the next statement to be executed. *Step into* steps into (imagine that!) the function and pauses execution at the beginning of the called function. *Step over* executes the entire function as a single statement and advances to the next statement after the function call. *Step over* is a huge time-saver once you know that you don't care about stepping through that function line-by-line. However if you want/need to see what's going on in the function, by all means step into it. *Step out* is what's used once you're in a function and you no longer care about what's going on inside the rest of it and just want to get back to the code that called the function. *Step Over* completes executing the function and gets you back to where you were when you called it. These three icons are shown below.

Just to get the hang of single-stepping, we're going to start with *Step Into*. You can do this (and most of the simulator functions) one of three ways. In decreasing order of pain, they are: in the menu bar do *Debug->Step Into*, in the toolbar click on the *Step Into* icon (the one immediately to the right of the yellow arrow in the toolbar, or by pressing the **F11** key. Learn to know and love the **F11** (and **F10**, which is coming-up later) key, as the function keys are by far the easiest ones to use 500 times in an hour.

Press the **F11** key once. It should now point to the statement where **buttonPresses** is initialized.



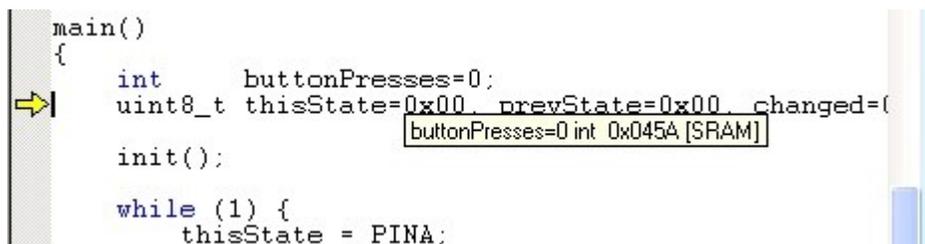
```

int
main()
{
  int    buttonPresses=0;
  uint8_t thisState=0x00, prevState=0x00, changed=(
  tz
  init();

```

Note again that this is the *next* statement to be executed. Press the **F11** key again and you should see the arrow pointing to

uint8_t thisState, prevState=0x00, changed;



```

main()
{
  int    buttonPresses=0;
  uint8_t thisState=0x00, prevState=0x00, changed=(
  buttonPresses=0 int 0x045A [SRAM]
  init();
  while (1) {
    thisState = PINA;

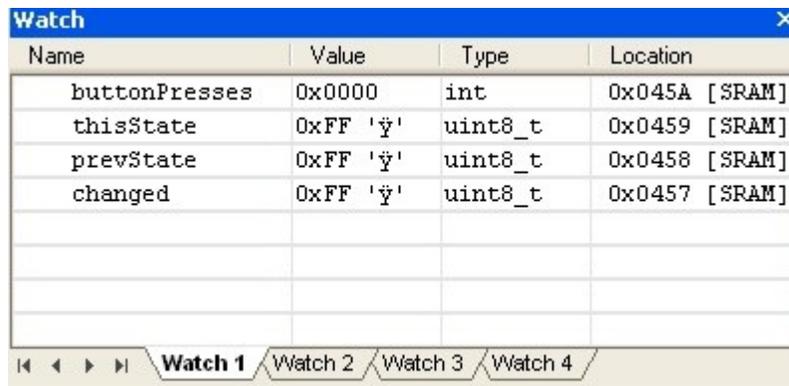
```

Illustration 11: Program execution location after first single-step

At this point, wouldn't it be cool to know that the value of **buttonPresses** is actually zero? Well, you can, and there are two ways to do this. The first is to use your cursor to point to the variable **buttonPresses** anywhere in the source window and it'll pop-up a floating balloon that says that the current value is **0** (along with some other details about the variable). The second way is the power-user way, and you're also going to learn that now.

You can maintain a window on the screen containing any variables that you want to watch along with their current values. This window is called the *Watch window*, and you can open it by doing a *View->Watch*. Do this and right-click the first cell under the label *Name* and click *Add Item*. If you enter **buttonPresses**, it will give you an entry showing the current name and value for this variable. Not only that, this value will be updated as you execute the program under your control (it isn't updated as the program free-runs though). An alternate way to add a new variable to be watched is to double-click in an empty *Name* field cell and you can enter a variable name that way. If that weren't enough, there's third way to add a variable to the watch window – just right-click on the variable name anywhere the editing window and select *Add Watch*. Pretty slick, huh?

While you are here, use any of these three methods to add watch entries for the variables **thisState**, **prevState** and **changed**. After this is done, the contents of the Watch window should appear like this:



Name	Value	Type	Location
buttonPresses	0x0000	int	0x045A [SRAM]
thisState	0xFF 'ÿ'	uint8_t	0x0459 [SRAM]
prevState	0xFF 'ÿ'	uint8_t	0x0458 [SRAM]
changed	0xFF 'ÿ'	uint8_t	0x0457 [SRAM]

Illustration 12: Watch window with four variables shown

Depending on your preference, (all, or selected) entries in the Value column can be shown as either decimal or hexadecimal number by right-clicking on the value in question.

You should notice at this point in program execution that we have not yet initialized these three variables. If you look where the yellow arrow is pointing, we're still before their initialization statements. Observe that these three variables currently contain some random, often non-zero, value.

They are currently what's known as “initialized to garbage”. This is programmer-speak for a variable that has been declared but not yet set (initialized) to any known value – it contains whatever garbage was there when that particular memory location was last used. It is a very, very, very bad thing to ever use a value that has not been initialized. Multi-billion dollar satellites (and perhaps your job with it) can be (and have been) lost forever when this happens, so it's usually best to try to avoid it.

Go ahead and press **F11** again to execute their initialization statements. Presto, Initialize-o... zero values now for all of them!

Before you go any farther you're going to learn how to view the I/O registers for the microcontroller. All **PORTA**-related registers can be viewed by expanding the **PORTA** group (click the plus-sign) in the right-hand *I/O View* window, as well as for all of the other functional groupings of I/O registers. Here you will find the current simulated values for **PORTA**, **DDRA** and **PINA** both there as well as in their own more-detailed-view window below. Go ahead and expand the **PORTB** set also. If you would like to add another group of registers (like those for **PORTB**) in the bottom detail window, you can CTRL-left-click on the register name and it will be added.

Now if you can't see the full contents of each of these entries' columns, especially in the lower of the two windows, stretch your windows out so that you can see everything. In particular, you want to be able to see all eight squares in the *Bits* column for each of the three **PORTA/PORTB**-related registers. These types of registers are very important as they are the chip's interface to the outside world.

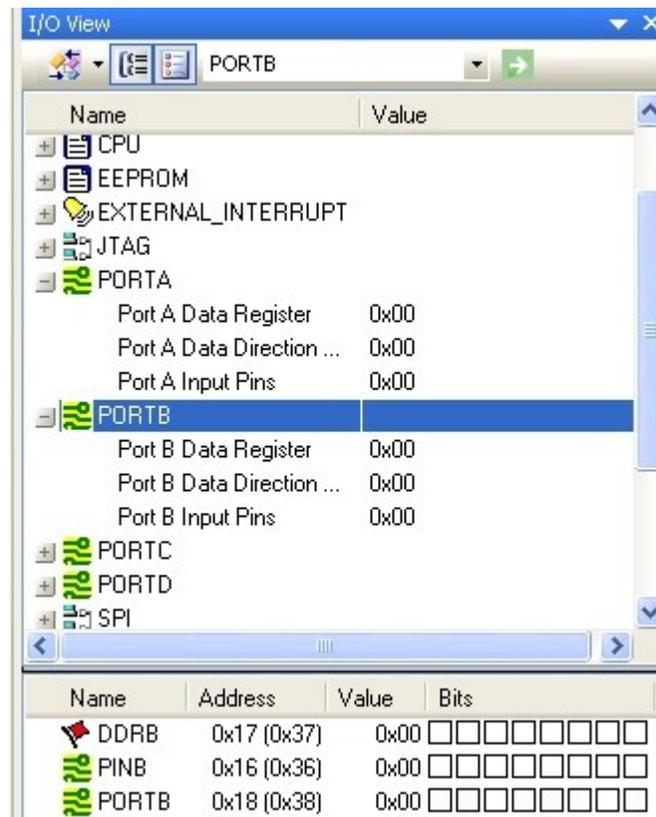


Illustration 13: Viewing current state of I/O registers

At this point, program execution should be right before the function call to `init()`. Let's demonstrate *Step Into* first. The first time that you hit **F11** you'll find yourself entering `init()`. Continue pressing **F11** until you have executed the following two lines:

```
DDRA = 0x00; // PORTA used entirely for input
DDRB = 0xff; // PORTB used entirely for output
```

Name	Address	Value	Bits
 DDRB	0x17 (0x37)	0xFF	<input checked="" type="checkbox"/>
 PINB	0x16 (0x36)	0x00	<input type="checkbox"/>
 PORTB	0x18 (0x38)	0x00	<input type="checkbox"/>

Illustration 14: Graphical result of setting DDRB bits to all Ones

You should notice that the values for DDRA and DDRB (also known as Port A/B Data direction register) in the *I/O View* window have been updated. Especially note that the hollow squares in the lower window are blackened for bits that contain high (1) values. This representation is very useful for quickly visualizing which bits are set within an 8-bit value. In the case of **DDRB**, all bits are high/black – assuming that you have the **PORTB** group selected in the upper window. You will often be using these windows frequently when following your code's execution.

At this point you should be done executing the last statement in **init()** and ready to return. Using **F11** will return you to the next statement after the one that called this function.

Since we've missed our only opportunity to demonstrate the use of *Step Out* (because there's only one function call in our demonstration program, and we're never calling it again), let's re-run the program from the beginning. You'll do this often when you get to a point where you've missed something that you mindlessly blew-over and need to start over again. Unfortunately, few simulators are able to back-up through a simulation, so you'll have to start at the beginning again.

To halt execution and start over again, press the *Stop* icon in the menu bar, the blue square (or *Debug->Stop Debugging*). Go ahead and do this now.



*Illustration 15:
Mr Stop
icon*

To restart, as before, press the green triangle (Start/Continue) to continue and halt at the first executable statement. Press **F11** until you get just inside **init()**. Rather than stepping through the rest of the function that you no longer care about, you can use *Step Out* to leave this function. As usual, you can use the *Step Out* icon to do this, *Debug->Step Out*, or the appropriate function key.

To demonstrate the final stepping control, do a *Debug->Stop Debugging*, then *Debug->Start Debugging* to start the program again, and **F11** until you are just about to call `init()` again. Now use *Step Over* to execute `init()` without bothering to step into it. You'll do this often once you are comfortable with what a function does and no longer care about crawling through it again. Stepping into functions that you don't care about is often a huge waste of time. Even better, notice that **F10** is the keyboard shortcut for *Step Over*.

F10 is even more useful than **F11** in many circumstances. To be honest, I never remember which hotkey is which. I'll either use the menu or the toolbar to pick which one that I want the first time, look at the FunctionKey label that is hinted as I hover over the icon, then just use the FunctionKey from that point on. Just trust me and use the keys – repetitive-stress injuries aren't pretty and every mouse click adds up, trust me.

By the way, one thing that's easy to do is to keep on pressing **F11** and then end up in a function you didn't intend to enter. In that case, just do a *Step Out* and you're back to where you wanted to be.

In summary, those are the three flavors of single-stepping and how each of them are useful, or more efficient, depending on how deliberate you need to be when stepping through code.

Viewing/changing Variables' and I/O Registers' State

At this point you should have just entered the `while` loop. This loop is coded to run forever until `PINA` is read and has the value `PINA_END_VALUE` (all buttons pressed), at which time it breaks out of the loop. Otherwise, every time through the loop the current contents of `PINA` are compared to the previous value to see if any bits have changed, indicating that a button has changed state. Press **F10** a couple dozen times and see if you can figure out what the program is doing so far.

It should be eventually become obvious that nothing's going to change because, well, nothing's changing. Clearly there aren't any buttons connected our simulated ATmega16 on `PINA/PORTA`, and until that happens, you'll be stuck only executing the boring part of the loop forever. However, a key feature of the simulator is that you can manually change values on-the-fly to get the program to behave like physical input changes have occurred on the chip's digital inputs.

First, let's change the variable `thisState` which contains a copy of the current contents of `PINA`. Open the *Watch* window that was introduced before if it's no longer around. If the variable `thisState` is no longer being watched for some reason, add it again as was done above. Single-step with **F10** until you get to the following statement:

```
if (thisState == PINA_END_VALUE)
```

So far it's been the case that **thisState** has always been zero, causing you to do little in the rest of the loop and to return to the top again. However, let's force a change. Use the *Watch* window to change the value of **thisState** from **0x00** to **0x01**. You do this by double-clicking on the **0x00** value and entering **0x01** in its place (and pressing **ENTER/RETURN**). Now, with a non-zero value for **thisState** at this point, pressing **F10** should eventually get us into the **if** block that we've never been in before. The first press of **F10** should get you past the **break** statement, as usual. Another **F10** will cause **changed** to be set to the difference between the current value and the value from the previous time through the loop. **Changed** becomes **0x01** because the lower bit has changed since the last time, which then gets you to the inner **for** loop. By doing enough **F10**s and following along carefully, you should see that a button press is detected, and **buttonPresses** is incremented to **1**.

If you continue **F10**-ing enough you'll find yourself back in the outer loop. Assuming that you don't manually alter the value of **thisState** as we did the last time, **PINA** will be read again as **0**, transferred into **thisState**, and it'll look like the low bit has changed back to **0**, signifying that the button has been released. You'll head into the inner **for** loop, nothing will appear as a new key *press*, and eventually you'll be back to where you started, everything zero, nothing changing, looping in the outer part of the **while** loop forever.

Interesting things happened along the way though. **buttonPresses** is now set to one because we've “seen” a button press. And if you look in the *I/O View* window, you should notice that the value of **PORTB** has changed to reflect the running count of presses detected. If you click on **PORTB** in the upper window, you'll see the detailed view of the **PORTB**-related registers in the lower window. Since **PORTB** would be connected to LEDs in our “real” hardware, one of them would now be lit-up, just like the blackened-square in the lower-right window, in the **0x01** bit position.

Name	Address	Value	Bits
DDRB	0x17 (0x37)	0xFF	■ ■ ■ ■ ■ ■ ■ ■
PINB	0x16 (0x36)	0x00	□ □ □ □ □ □ □ □
PORTB	0x18 (0x38)	0x01	□ □ □ □ □ □ □ ■

Illustration 16: I/O Register graphic showing single-bit change

Another important feature in addition to being able to see variables' values at every execution stop-point is that you can also change the I/O registers' values on-the-fly. Just as you manually changed `thisState` to modify the captured value of `PINA`, you can modify `PINA` directly in a similar fashion. Go ahead and click on `PORTA` in the upper I/O View window. `PINA`'s bits in the lower window should all be zero – clear squares in the *Bits* display. Now click on one of the clear squares for `PINA`, and you'll see it switch to black. You've turned this bit on manually, and you'll notice that the displayed numerical value changes with it, highlighted in red. This red high-lighting is very helpful feature also, and it's used to point out what value was most recently changed on the display. Go ahead and press on all these squares to light them all up *except one*.

Name	Address	Value	Bits
DDRA	0x1A (0x3A)	0x00	<input type="checkbox"/>
PINA	0x19 (0x39)	0x7F	<input type="checkbox"/> <input checked="" type="checkbox"/>
PORTA	0x1B (0x3B)	0x00	<input type="checkbox"/>

Illustration 17: I/O Register graphic showing value after manual change

This would denote that all buttons except one are currently being pressed/held at the same time. Now if you go back and step through the `while` and `for` loops, you'll see these additional (fake) button presses counted, eventually resulting in a `buttonPresses` value of `8`, and the binary representation for eight displayed on `PORTB`. And just as you can simulate button presses, you can simulate releases by clearing the bits in the display for `PINA` also. If you set all eight `PINA` bits at the same time, that's the magic sequence that causes the program to break-out of the outermost `while` loop and essentially end the program. All buttons pressed results in `PINA` being `0xFF`, to which we've associated the constant label `PINA_END_VALUE`.

So in summary, you've seen in this section that you're able to modify C-language variables on-the-fly in the *Watch* window, and able to modify `PORT` input pins on-the-fly, and can view their output state in the *I/O View* window.

Project Display Mode vs. Debugging Display Mode

If you haven't noticed already, AVR Studio pops back-and-forth between two modes while you're using it. As soon as you hit *Start Debugging*, the left-hand window switches to a *Processor* window. When you hit *Stop Debugging*, it reverts to its starting (editing) state and shows you the *AVR GCC* window, which is basically a project view and the files contained therein.

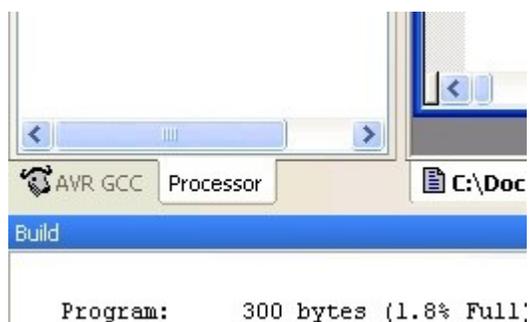


Illustration 18: The magic tab to recover the Processor values window

The point here being that if you're expecting or wanting to see the project view in the left-hand window and it's not there at the time, you can switch to it by hitting the window tab for it *below* the window.

Using a Stimulus File for Playing-back a Sequence of Inputs

It's clearly very convenient that you can fake I/O input in the simulator by manually setting/clearing bits in **PORT** registers, but what's even more convenient is that you can automate this process. By specifying a time-line of what you want the input pins to be, the AVR Studio Simulator will change them for you automatically at specified points during the execution of your program. The mechanism for doing this is by providing a *stimulus file* for any **PORT** bits that you have configured for input. You've already got one of these files – it's the **simcounter-PORTA.sti** file that you cut-and-pasted at the beginning of this exercise.

We'll take a look at this file. You can use the Windows FileManager to find it, but you can also get to it in the *Other Files* section of the project display window (*AVR GCC*). Regardless of you how want to look at it, get it open now so we can talk about its contents.

```
...
00013000:00
00014000:01
00015000:03
00016000:01
...
```

A stimulus file is a two-column data file. The first column is basically a time designation (it's actually the Cycle Count value that shows up in the Processor window on the left-hand side of the screen) and the second column is the value that you'd like the **PIN** register to be at that point in time. The last value in the file must be 99999999, according to the documentation. One stimulus file is good for only one **PORT/PIN** eight-bit register, although multiple log/stimulus files can be used to get access to the other ports. In our case all of our input is on **PORTA/PINA**, so we only need one file.

To attach a stimulus file to the simulator, go to *Debug->AVR Simulator Options->Stimuli and Logging*. I've noticed that if you haven't started a simulator run (*Start Debugging*), the AVR Simulator Options option may be grayed-out, just like that other goofy case you saw above. If so, start a simulator session and at the first pause, pop open that config panel. Choose *Port PORTA*, *Function Stimuli* and use the "...” button to set *File* to the location where **simcounter-PORTA.sti** is located. When you click *Add Entry*, you should see the new entry show up in *Action List*. Finally, press **OK**.

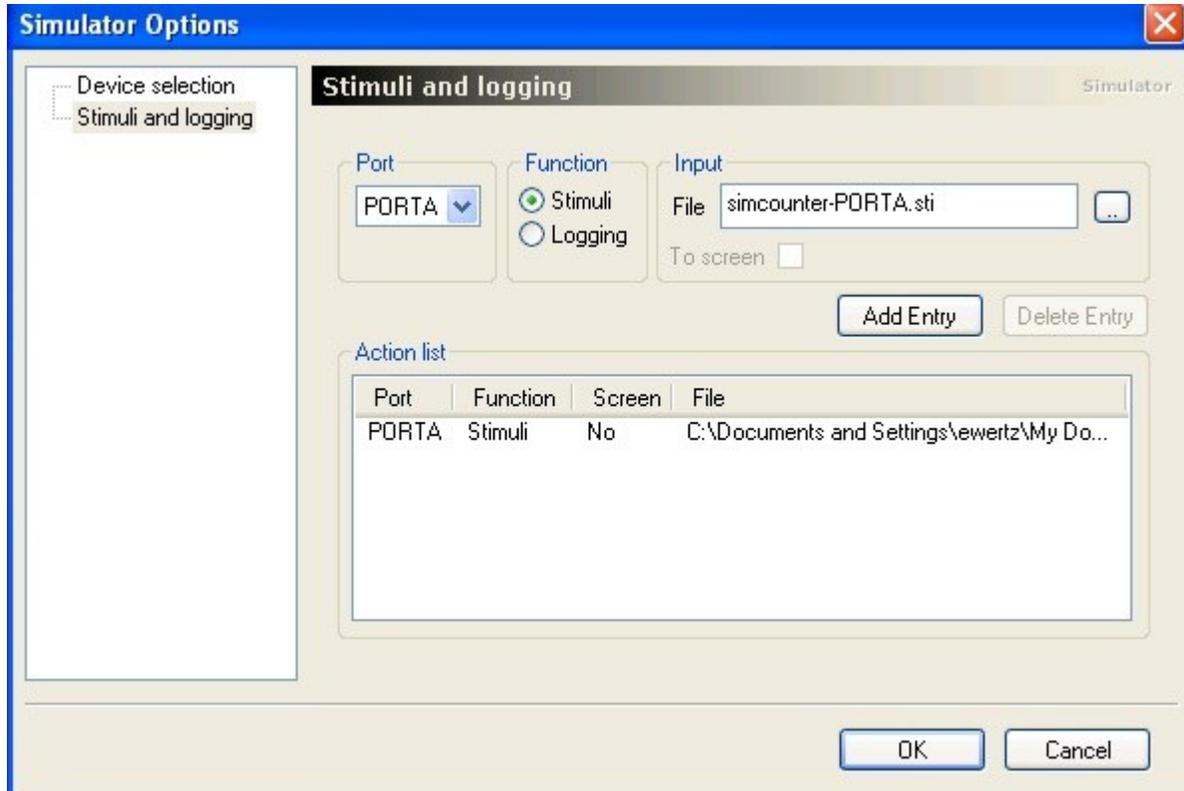


Illustration 19: Choosing PORT and filename settings for I/O Stimulus/Logging

Similarly, if you'd like to log the output of **PORTB**, you can do so by selecting Logging and by providing a filename for output. You don't necessarily have to do this now, but I mention it just to let you know that exists. It might even be possible get Excel to import both the input stimulus file and output log file and plot them on the timescale so that you can compare their interrelationship, just like a two-channel oscilloscope would do.

Use of Breakpoints

Now that you've got the stimulus file in place, let's start running again. Once again, do a *Start Debugging* (*Stop Debugging* first if you need to), and keep banging away on **F10** until something interesting happens. OK, don't bother anymore after you've done this for about 20 seconds because it's going to be a while until something happens. The point here is that in real life, most microprocessors execute so quickly that even if you were to press a button right away after your hardware starts running your program, you're really going through the while loop hundreds of thousands of times before **changed** will ever become non-zero. This isn't because it takes the microcontroller so long to figure out that you've pressed the button, it's because it's so incredibly fast that it executes those five statements just inside the **while** loop approximately 200,000 times per second. So if you were to simulate the program waiting for a button press that occurred one second after the start of execution, you'd have to press **F10** about (literally) a million times before you ever saw that inner **for** loop being hit. Borrrrrrrrr-ing!

Fortunately there's a feature (in almost every debugger) called *breakpoints* that tell the simulator (or debugger) to just go ahead and run full-speed until it hits a particular place in the code. When a breakpoint is hit, the program stops execution and passes control back you to so that you can either start single-stepping, look at variables, or do whatever else you want to do.

Since the point in the program where things are most interesting is around when a button press has been detected, let's set a breakpoint there. Within the source code window there is a vertical gray bar along the left side that is used to select actions or to view settings on a line-by-line basis. Within this gray bar, right-click on the same line as the statement:

```
buttonPresses++;
```

and select *Toggle breakpoint*.

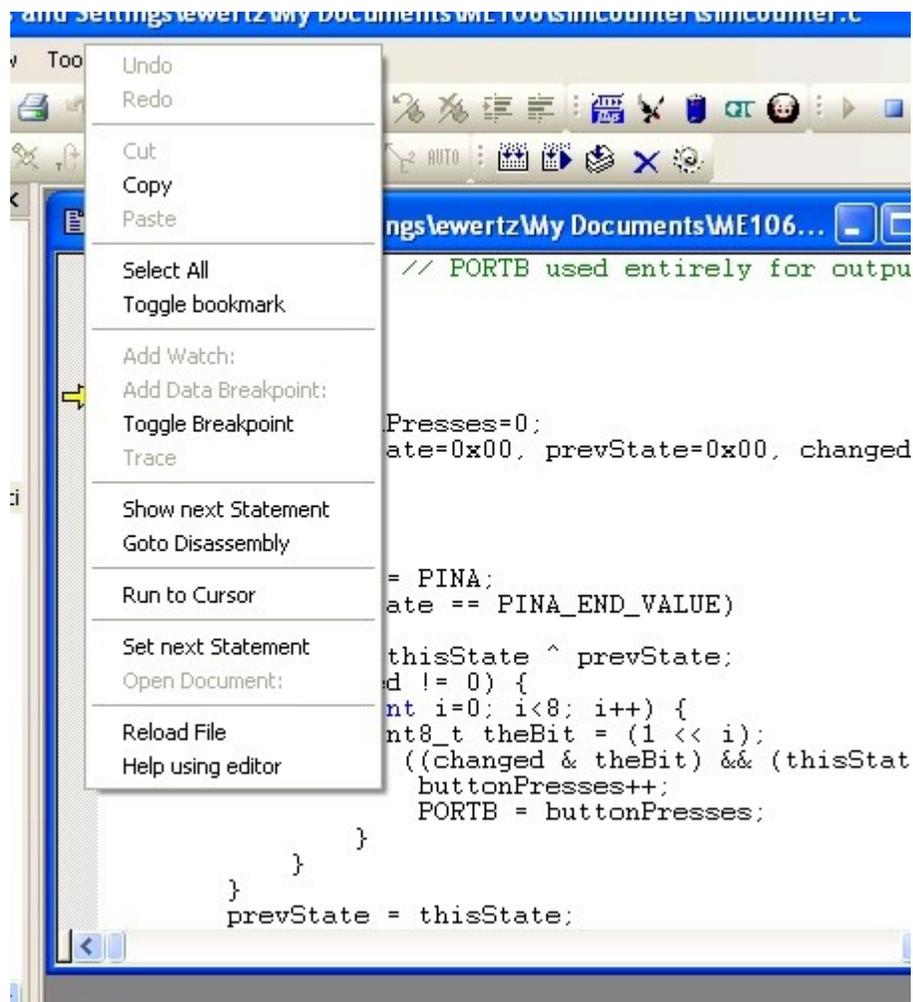
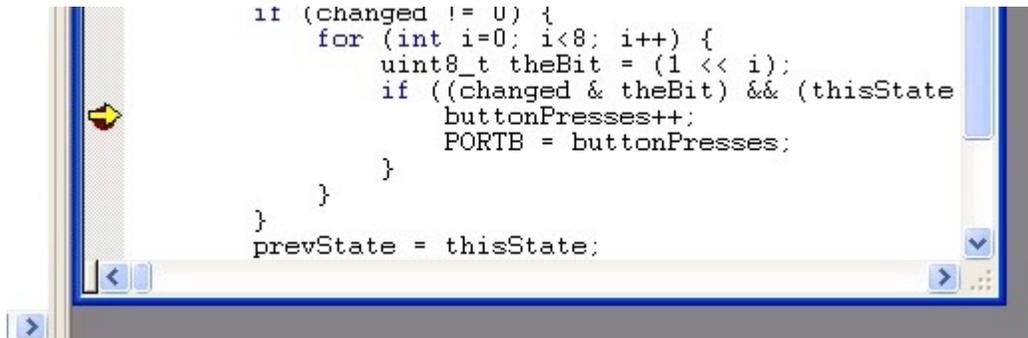


Illustration 20: Setting/toggling a breakpoint by right-clicking on the gray line-select bar

The operative word here is *toggle*. If there's a breakpoint there already, it removes it, and if there's none there it'll add one. Once you've got as many breakpoints as you want set, you can let the program free-run until it hits one of them. In our case, there's exactly one. To let the program run wild until it hits a breakpoint (or ends), do a *Debug->Run*, press the *Run* icon, or **F5**.



```

if (changed != 0) {
    for (int i=0; i<8; i++) {
        uint8_t theBit = (1 << i);
        if ((changed & theBit) && (thisState
            buttonPresses++;
            PORTB = buttonPresses;
        }
    }
}
prevState = thisState;

```

Illustration 21: Execution stopped when breakpoint hit

Once you hit the breakpoint, the state of the simulation is frozen. You have access to all the current program variables as well as the PORT pins. From here you can often tell exactly what set of conditions were that got you to this breakpoint.

Breakpoints are an incredibly useful feature. Not only are they huge time-savers, but they're remarkably efficient for detecting error conditions – assuming you write your code to try to detect them in the first place. All you have to do is to put breakpoints everywhere where you don't expect your code to ever end-up. If you ever hit one of these breakpoints, you'll be stopped exactly where you want to be to track down the problem.

You want spots in your code to detect situations like this. However instead of having a breakpoint for it in the debugger, you'd have the program set off an alarm, hit a kill-switch, or perhaps even reset itself. Otherwise, you might just be kissing that satellite goodbye *without even knowing why*.

Simulator Limitations

The most notable limitation of the AVR Studio Simulator is that although it does good job of simulating digital input, analog input simulation is not implemented. This means that you cannot use a stimulus file to feed analog input values into an ADC pin (you'll get to analog pins and ADCs later in the semester). However, this limitation can be worked-around by writing your code so that you have the opportunity to provide your own fake ADC values (to replace the values returned by the registers that aren't simulated correctly) using the Watch window. Or better yet, internally/automatically generate fake ADC values in code that's only enabled during simulation, and you can probably make a fair amount of forward progress that way.

Wrap-up

Hopefully by now you're convinced that there's a lot of software work that can be in (and before!) the lab before the circuit's been wired-up. In most of the labs this semester, the software can almost be complete by the time that you're ready to flip the power-switch on real hardware.

There's no reason why there has to be only one “software person” per group – as there often has been in semesters past. Everyone has the ability to show up to lab with test software ready to run and debug. This is especially important because typically roughly a third of the lab groups don't finish half the labs in the allotted time, so any head-start will go a long way toward finishing on-time with less stress.

There are plenty of other features of AVR Studio that we haven't discussed here. There is both reasonably complete built-in help available as well as a fairly well-organized set of menus and panels to poke around in. There is also a fair amount of customization that you can do if you don't like the default layout of the windows within the main frame of the application. There's a lot of stuff in AVR Studio, so if you find yourself spending all day with it, you might as well make it comfortable by customizing it.

So enjoy the AVR Studio Simulator. It contains all (ok, *some*) the fun of embedded programming without any of that nasty short-circuited hardware smell. And it's all free... sweet!

ADDITIONAL INFORMATION

Other AVR Simulator references

AVR Studio has a fair amount of embedded HTML help built-in. It's a very concise and convenient source of information on simulator features. To the best of my knowledge, there's no one PDF document from Atmel that describes the use of the AVR Studio Simulator.

The only other document that I have found that discusses the use of the AVR Studio Simulator in any detail is

http://www2.tech.purdue.edu/ecet/courses/ecet309/Reference_Materials/Simulation_AVR_Studio_4.pdf. Although the document makes many references to their own development environment and hardware, there's a lot of useful material contained therein.

Code optimization

The standard reference documentation for GCC (the compiler) optimization options may be found here: <http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/Optimize-Options.html#Optimize-Options>

In short, optimizations are transformations that the compiler can do that can affect either the amount of code that it takes, or the time that it takes, to execute your program. In some cases you may need (or want) to choose between smaller size and faster speed of your final program – in the former case if your program simply won't fit into the memory of your MCU, or in the latter case if the time (or power) to execute it should be minimized. In many circumstances, these two desires can be mutually exclusive where you're either at the space or time limits of your requirements.

Optimization of code by the compiler can have some tricky consequences. Let's just say that there are many “reasonable” or “smart” things that the compiler can do when generating object files that can bite you in two ways – first, it does something that you're not expecting it to do but you can't really fault it for, or second, it does something that is reasonable with your marginal, ambiguous, or *incorrect* code that sometimes you can get away with, and sometimes you can't.

A simple example that demonstrates some of the effects of optimization is the following:

```
main()
{
    int a;

    motor_start();

    a = 1;
    for (a=0; a<1000; a++)
        ; // do nothing, just loop 1000 times

    motor_stop();

    printf("%d", a);
}
```

If you tell the compiler to apply no optimizations to this code (compiler option `-O0`), it compiler will generate code pretty much exactly as you'd expect it to – call `motor_start`, `a=1`, `a=0`, `a=1`, `a=2`,...,`a=999`, call `motor_stop()`, and print “1000”. If you single-step through this code in the simulator, it will stop at pretty much every line where there's a code statement, including 1000 times through the loop (boring!).

However, once you tell the compiler that it's allowed to make some minimal (time or space) optimizations, it might choose to do something like eliminating the statement “`a = 1;`”. It can reasonably do this because it can look at the code and say “look, why is this guy assigning `a=1` when `a` is immediately going to be set to zero in the very next statement?”. It can simply eliminate that statement, and *it will*. Now, if you were to simulate this code at this optimization level, you might single-step up to `a=1` and have it skip right over it, *because the compiler has completely eliminated this line*.

However, it gets even better. If you crank-up the optimization level to something even higher, the compiler looks at the code even closer and says, “look, why is this guy wasting the time to use the variable `a` to count to 1000 *and do nothing with that value* -- when I know that at the end `a=1000`? So what I'll do is to just replace the whole loop with `a=1000;` and be done with it!”. If you were to simulate this program with this level of optimization (perhaps, `-O2`), you would single-step and it would stop nowhere but `motor_start()`, `motor_stop()`, `printf()`, and at some line before that where it inserted “`a=1000;`”. This is one of the symptoms that I warned you about above – unless you specify `-O0` for the optimization option, single-stepping may *appear* to act bizarre.

However, it gets even better. If you were counting on the time that it took to count to 1000 to have that motor spin, it ain't gonna happen because the loop is gone. The code would have to be changed to preserve the required delay time, and this can be done a number of ways which you'll learn about later. The error was to assume that a certain type of code would be generated, and this is always a dangerous assumption. Sometimes you can skate by without taking the time to fully understand C or how one's own particular compiler works and merely turn off all optimizations. However, in the end, you've got buggy code that's going to break sooner or later (if it ever works at all), and that can be a dangerous thing.

These are just a few of the things that the optimizer can do if you tell it that it can be smarter. Because the end result of the program is the same as if it hadn't done so, these are all technically correct transformations for it to do. The optimizer in the compiler is more often your friend than your enemy once you start developing serious code. However, understanding how the C language really works to produce correct and efficient code isn't something that you're going to learn without some substantial effort.

SUPPLEMENTAL FILES

simcounter.c:

```
/*
 * simcounter.c
 *
 * This program is primarily intended to be used in the AVR Studio
 * Simulator for demonstration of debugging features.
 * This program counts button presses on eight buttons on PORTA,
 * and continuously displays the current count on eight LEDs
 * on PORTB.
 *
 * By going through the exercises that are paired with this program,
 * you will become familiar with the AVR Studio Simulator and
 * Debugger.
 */
#include <avr/io.h>

#define PINA_END_VALUE 0xFF

void
init()
{
    DDRA = 0x00;    // PORTA used entirely for input
    DDRB = 0xff;    // PORTB used entirely for output
}

int
main()
{
    int    buttonPresses=0;
    uint8_t thisState=0x00, prevState=0x00, changed=0x00;
```

SJSU ME106

```
init();

while (1) {
    thisState = PINA;
    if (thisState == PINA_END_VALUE)
        break;
    changed = thisState ^ prevState;
    if (changed != 0) {
        for (int i=0; i<8; i++) {
            uint8_t theBit = (1 << i);
            if ((changed & theBit) && (thisState & theBit)) {
                buttonPresses++;
                PORTB = buttonPresses;
            }
        }
        prevState = thisState;
    }

    PORTB = buttonPresses; // just a place to put a final breakpoint
    while (1)
        ;
    /*NOTREACHED*/
}
```

simcounter-PORTA.sti:

```
00000000:00
00010000:01
00011000:00
00012000:01
00013000:00
00014000:01
00015000:03
00016000:01
00017000:03
00018000:01
00019000:00
00020000:0F
00021000:00
00022000:03
00023000:02
00024000:01
00025000:00
00100000:FF
99999999:00
```