Lab Assignment 2—In the Blink of an Eye
Assembly Language Programming on the ReadyAVR

Overview:

In this lab you will analyze, modify and test an LED "blink" program on the ReadyAVR microcontroller board. The lab has three objectives:
1. become familiar with the AVR Studio development environment, including the debugger,
2. determine the ATmega128A's clock speed by analyzing an assembly language program, and
3. modify the program to change the LED's duty cycle and blink rate.

You will also write a lab report describing the work you've done, including answers to questions posed in this assignment sheet.

Lab Procedure:

If you haven't already done so, accept the Lab02 assignment invitation you received from your instructor. This will create a remote repo pre-populated with starter code. Once the remote repo is created, open it in **GitHub** and use the Code button. Cmd line or GitHubDesktop are both acceptable.

Place this new repo under your existing directory. (`D:\EECE3624Labs, or C:\*`).This should be under the Atmel Studio solution directory made in the last lab, and next to the Lab01 repo from last week. For the command line option, you need something like this:

```
D:\EECE3624Labs> git clone <remote-repo-URL> Lab02
```

Open up the EECE3624 solution in **Atmel Studio's Solution Explorer** (Double-click on the `EECE3624Labs`).   In the File menu, click **Add | Existing Project…**, browse to `D:\EECE3624Labs\Lab02` and select the `Lab02.asmproj` file. This will add the `Lab02` project to the `EECE3624Labs` solution.

Set the overall solution to default to Lab02 for :
- R-click on the newly added `Lab02` project and select **Set as StartUp Project**.
- Then R-click on the `Lab02.asm` assembly file and select **Set as EntryFile**.
  Remember, if you don't do this for each new project you'll end up compiling and running previous lab code!

Connect the Atmel-ICE to the ReadyAVR board and then connect both devices to your computer. In Atmel Studio, double-click on the `Lab02.asm` filename in Solution Explorer to open the file for editing. Click on **Project | Lab02 Properties**… and choose **Tool** from the left-hand menu. Select your Atmel-ICE device as the debugger/programmer, and set the Interface to JTAG. Finally, select **File | Save All**… to save any unsaved files.

Click the green ▶ triangle on the AVR Studio menu bar to assemble, download, and run the code on the ReadyAVR. LED3 should begin blinking. The percent of time LED3 is "on" compared to the total "on-plus-off" time is called its **duty cycle**. In this case, LED3 has a 50% duty cycle (i.e. on and off times are equal). Click the red ■ button to stop the debugger. Now,

using what you know about duty cycle, answer the following question **in your lab report**:

Question 1: What is the duty cycle if an LED is on for 0.3 sec and off for 0.5 sec? Show your work including proper units!

Next, you'll practice using the Atmel Studio debugger. This is a **powerful tool** which can make your job as a developer **much, much easier**. It is well worth your time to learn how to use it!

Open the Debug I/O View by selecting **Debug | Windows | I/O** from the top menu. You can choose to dock or undock this window (I prefer to dock it to the right side of Atmel Studio). Click the left margin of the code to set a *breakpoint* at the `ldi R16,(1<<DDC3)` instruction. Display the I/O View and PORTC configuration map as shown in Figure 1. **Make your display look as much like this as possible**.
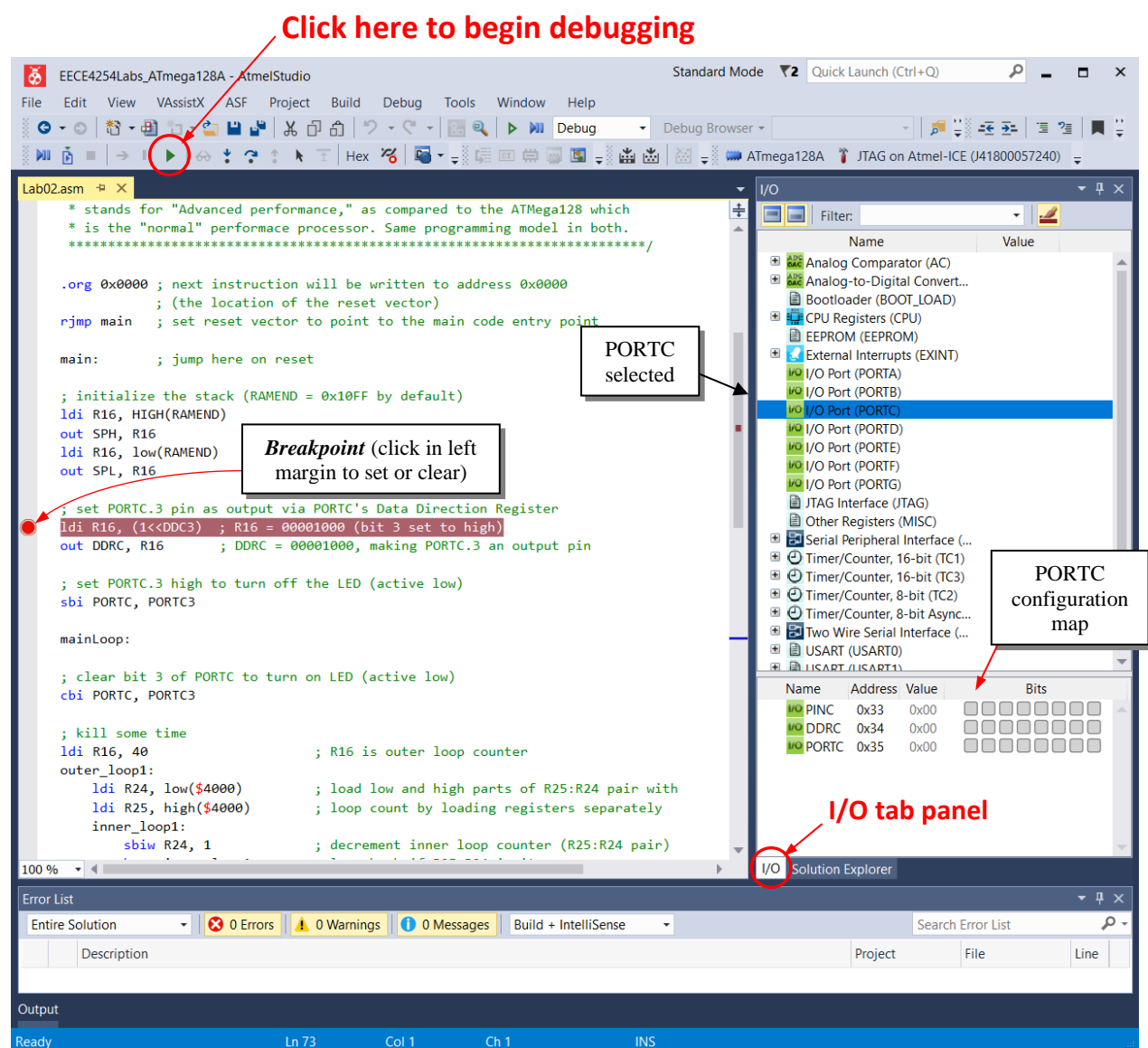
Figure 1. Atmel Studio showing the Debug I/O View and one breakpoint.

Click on the small green arrow on the top menu bar (see Figure 1 above) to begin debugging. The code should load into the ReadyAVR and begin running, <u>but it will pause at the breakpoint</u>. The line with the breakpoint will **turn yellow** and will have a **yellow arrow** at the left. This is shown in Figure 2 below. Note that when the debugger pauses on a particular line, <u>**the instruction on that particular line has *not* yet executed**</u>! This is a very important point! Click **Debug | Windows | Processor Status** to open the Processor Status panel (dockable, see Figure 2). It shows the current values of the Program Counter, the Stack Pointer, the X-, Y-, and Z Registers, the Status Register, and all 32 of the general purpose registers (R0 – R31).
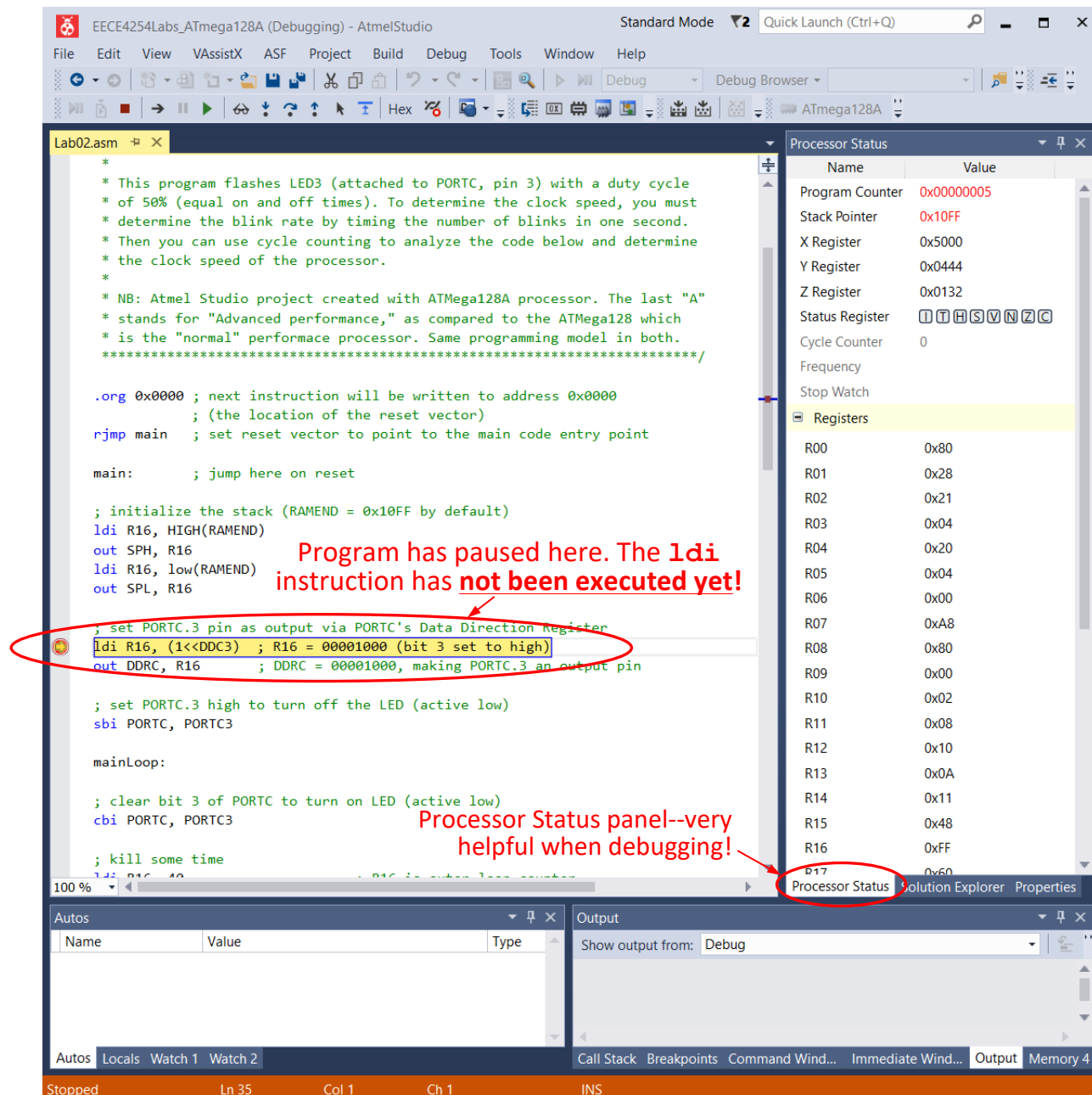

**Figure 2**. Debugger stopped at a breakpoint.

Notice the Program Counter in Figure 2 (upper right) has a value of 0x00000005, meaning <u>the next instruction to be executed is stored in program memory at address 0x00000005</u>. Does this make sense? There are many, many lines before the breakpoint, but **only five of them are actual program code**! So yes, the display is correct. The sixth instruction (at program memory address 0x00000005, counting from zero) is highlighted and *ready* to be executed, but hasn't actually executed yet. If this is not perfectly clear to you, **ask your instructor for clarification**.

Also notice that the General Registers are visible (if necessary, click the "+" symbol next to the word "Registers" in the Processor Status panel to see each register's value). Currently R16 contains the value 0xFF. **Look at the code and make sure you see why this is true**. Next, we'll **single-step** the processor and observe what happens when the next instruction is executed. Do this by selecting "Debug" from the top menu and clicking "Step Over" <u>once</u> to execute only the next instruction. After you've done this, answer the following questions **in your lab report**:

Question 2: What is the value of the Program Counter (in hex)?

Question 3: What is the value in register R16 (in hex), and why is the value shown in **red**?

Click "Step Over" once more (or use the **shortcut key F10**) to execute the next instruction (out DDRC, R16). Then click on the I/O panel next to the Processor Status panel, click on PORTC in I/O 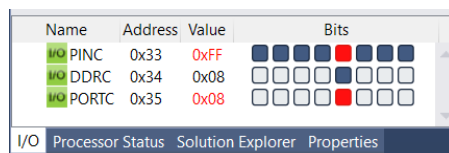View, and look at the PORTC configuration map (shown below). **Solid** color (**Red** or **Blue**) means the bit is <u>set</u>. **Red** means the change JUST happened.
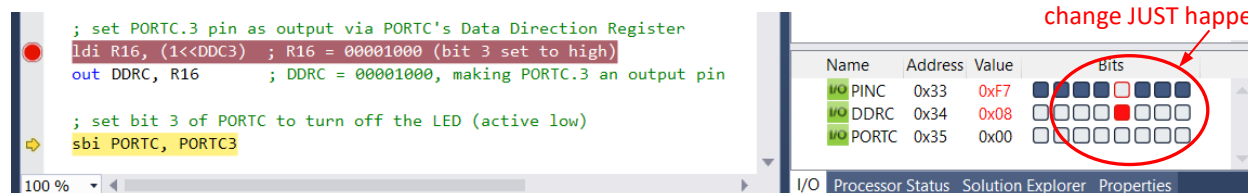
Figure 3. PORTC configuration map after executing the out DDRC, R16 instruction.

The configuration map shows DDRC's bit 3 is **set** (i.e. has a value of 1). <u>Setting a bit in a DDRx register makes the associated PORTx pin an **output pin**</u> (i.e. it can drive external devices). Any DDRx register bit that is clear (i.e. has a value of "0") makes the associated PORTx pin an **input pin** which can be used to read external signals such as switch positions or logic levels. **This is an extremely important concept. Be sure you understand what's going on here. <u>If not, ask!</u>**

Press F10 one more time to execute the sbi PORTC, PORTC3 instruction. Since bit 3 of PORTC was previously configured as an output pin (by setting bit 3 of the DDRC register), setting bit 3 of the PORTC register causes a voltage high (a logic "1") to appear on the bit 3 pin of the PINC register. The PORTC configuration map should look similar to Figure 4 below **and LED3 should be off** (active low). If this is not the case, go back to Figure 1 and repeat this procedure. If you're still unable to get to the point where LED3 is off, **ask your instructor for help**, but only after you've tried this a couple of times yourself.

Figure 4. PORTC configuration map after executing the sbi PORTC, PORTC3 instruction.

Press F10 one more time to execute the `cbi PORTC, PORTC3` instruction, turning LED3 on.

Next, look at the **outer_loop1:** section of the code. It creates a time delay by re-executing a group of instructions over and over again. If you study the code you'll see there are two loops— an outer loop and an inner loop. Together these "waste" millions of clock cycles. If you were to manually single-step through this code it would take more than a **week** to finish both loops! Since you don't have weeks to waste stepping through this code, let's use the "**Run to Cursor**" feature instead. R-click on the `sbi PORTC, PORTC3` instruction (about 10 lines below the current debug line) and select "Run to Cursor." This will cause the processor to execute instructions at **full speed** until reaching the line where the cursor is. Once that line is reached, Atmel Studio will go back into single-step mode and wait for further input from you. Notice that **LED3 is still on**, but if you **press F10 once again**, the `sbi PORTC, PORTC3` instruction will execute and LED3 will turn off, and the configuration map will show PORTC bit 3 has been set.

Congratulations! You now know how to use the Atmel Studio debugger to examine register values, single-step through code, and run at full speed to any given point in your program.

Analysis:

The goal of this final section is to **estimate the clock speed of your ATmega128A processor**. First, run the `Lab02.asm` program and **count the number of times LED3 blinks in exactly one minute** (have someone time you while you count). Include this count in your lab report.

Next, look at the assembly language code below used to turn LED3 on and off. **Look up each instruction** in the Atmel 8-bit AVR Instruction Set document and determine how many clock cycles each one takes to execute (*NB:* **brne** can take longer depending on execution results!)

```
    mainLoop:

    ; clear bit 3 of PORTC to turn on LED (active low)
    cbi PORTC, PORTC3

    ; kill some time
    ldi R16, 40                     ; R16 is outer loop counter
    outer_loop1:
        ldi R24, low($4000)         ; load low and high parts of R25:R24 pair with
        ldi R25, high($4000)        ; loop count by loading registers separately
        inner_loop1:
            sbiw R24, 1             ; decrement inner loop counter (R25:R24 pair)
            brne inner_loop1        ; loop back if R25:R24 isn't zero
        dec R16                     ; decrement the outer loop counter (R16)
        brne outer_loop1            ; loop back if R16 isn't zero

    ; turn off LED (set bit 3 of PORTC)
    sbi PORTC, PORTC3

    ; kill some time
    ldi R16, 40                     ; R16 is outer loop counter
    outer_loop2:
        ldi R24, low($4000)         ; load low and high parts of R25:R24 pair with
        ldi R25, high($4000)        ; loop count by loading registers separately
        inner_loop2:
```

```
        sbiw R24, 1              ; decrement inner loop counter (R25:R24 pair)
        brne inner_loop2         ; loop back if R25:R24 isn't zero
    dec R16                      ; decrement the outer loop counter (R16)
    brne outer_loop2             ; loop back if R16 isn't zero

  ; play it again, Sam...
  rjmp mainLoop
```

You now have two very important pieces of information:

1. The number of times LED3 blinked in one minute, and
2. The number of clock cycles the code consumed to cause that blinking

Armed with these two pieces of information, **calculate your processor's clock speed**. In your lab report, **include the step-by-step calculations you performed** and **clearly indicate your predicted clock speed** (with proper units!) <u>**Show your work neatly and clearly**</u> in your lab report.

Digging Deeper:

Finally, modify the timing values in the delay loops to make LED3 **blink at a rate of 200mS** (i.e. 200mS from the start of one blink to the start of the next) with a **duty cycle of 20%** (ratio of "on" time to "on + off" time). Provide this timing information in your report, and include your modified delay code in an Appendix in your report. <u>Be sure to add your name and date to the header block as the person who modified the code</u>, and remember that appendices are always numbered sequentially ("Appendix A,", "Appendix B," etc.) and, in the case of multiple appendices, **each one starts on a new page**.

Report:

This is a **one week lab**. Write up a report that outlines what you've learned as a result of performing this lab. <u>A lab report template, the technical content grading rubric, and the report writing style rubric are all available on the class Google Drive share and have been included in the initial Lab02 repo</u>. Use them! And if that's not enough, Appendix A of this handout offers even more guidance on writing lab reports for this class.

Again, be sure to **answer all questions and discussion points raised in this lab**, and don't forget to include an **Appendix** containing your commented code for the "Digging Deeper" modification.

Submit a **PDF** version of your lab report via Canvas, and your modified blink code in the project's remote GitHub repo. <u>**Both must be submitted on or before the beginning of lab on Feb 5, 2021**</u>.

Appendix A
Guidance on Writing Lab Reports

1. Follow the **template**! That's why I gave it to you. (Font type, sizes, spacing, page numbers, figure references, spacing before/after figures and tables, etc., etc., etc.).

2. Write a lab report in past tense. You're describing what you did in the past!

3. Proofread. Rinse. Repeat.

4. Use the Writing Center. They can really help you!

5. Besides text, reports can only contain Figures and Tables. That's it! Sure you can include graphs and drawings, but those are figures. Figures and tables must be numbered (with integers), centered and labeled. Figure labels go <u>below</u> the figure. Table references go <u>above</u> the table. It's weird. It's also IEEE style. So just do it…

6. Figure- and table labels should be centered

7. All figures and tables must be referenced in the text. A figure or table is expensive in terms of report real estate, so only include them when they enhance/clarify/expand what the text says. Don't assume the reader can "figure out" what a figure or table is telling them. Also, position a figure or table **close to but *after*** the first textual reference to it. <u>Figures and tables that "magically appear" before the text even references them are confusing to the reader</u>.

8. Real estate is precious. Don't squander it! (e.g. don't use a table for only one or two or three values).

9. Use units on all physical quantities (time, volts, ohms, etc.)

10. Use a real schematic editor (Multisim is on lab machines; also free tools on-line)

11. Use standardized schematic symbols and include polarities, reference numbers and values (with units)

12. Engineers are not "sloppy" with data. Words like "about," "sort of," "near," and "about" just cause the reader to doubt your results

13. Don't use colloquial expressions (e.g. "messed around with," "a drop in the bucket," "a no-brainer," "super easy")

14. Conversational numbers less than 10 must be spelled (*"I ate 12 hot dogs in three minutes."*)

15. Align program comments whenever possible.

16. Don't allow code to "wrap around" creating distracting "overflows." Instead either
    a) use a smaller font size (don't go under 11 pt. in any case),
    b) print landscape, or
    c) manually reformat code to make comment lines fit page width

17. Be succinct

18. Use equation numbering at right, inside parenthesis

19. Write hex as 0x1234. Similarly, write binary as 0b0001_0010. Be consistent throughout your document.

20. Look up the IEEE formatting standards for technical papers. You can never go wrong using IEEE formatting when doing ECE labs!