Lab Assignment 6—*Si, Señor y Señorita! Si! Si!* ReadyAVR C Programming

Overview and Software Architecture:

This lab builds on what you already know about interrupts, but we're leaving assembly language behind and moving on to C. And while C is quite different from AVR assembly, by using the predefined `#include` files in Atmel Studio you'll be able to access the internal hardware features of the ATmega128A in a very intuitive and familiar way. The goal of this lab is to create several "flashing patterns" using eight LEDs attached to the ReadyAVR board, and to then use the on-board joystick's center pushbutton to switch between those patterns.

Preliminary Setup:

Your ReadyAVR may already have eight LEDs with pull-up resistors and connector pins soldered onto the board's prototyping area (see Figures 1 and 2). **If not, open `EECE3524Lab06-LEDInstall.pdf` file and follow the instructions to prepare the board.**

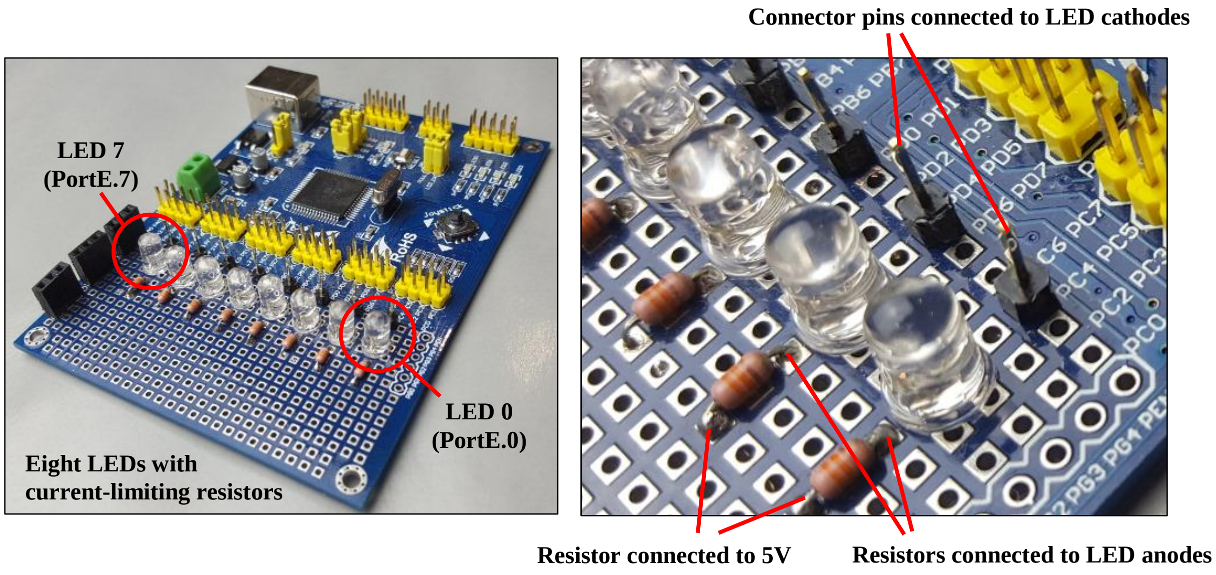
LED 7
(PortE.7)
LED 0
(PortE.0)
Eight LEDs with
current-limiting resistors
Connector pins connected to LED cathodes
Resistor connected to 5V
Resistors connected to LED anodes

Figure 1. ReadyAVR with active-low LEDs and 220Ω current-limiting resistors.

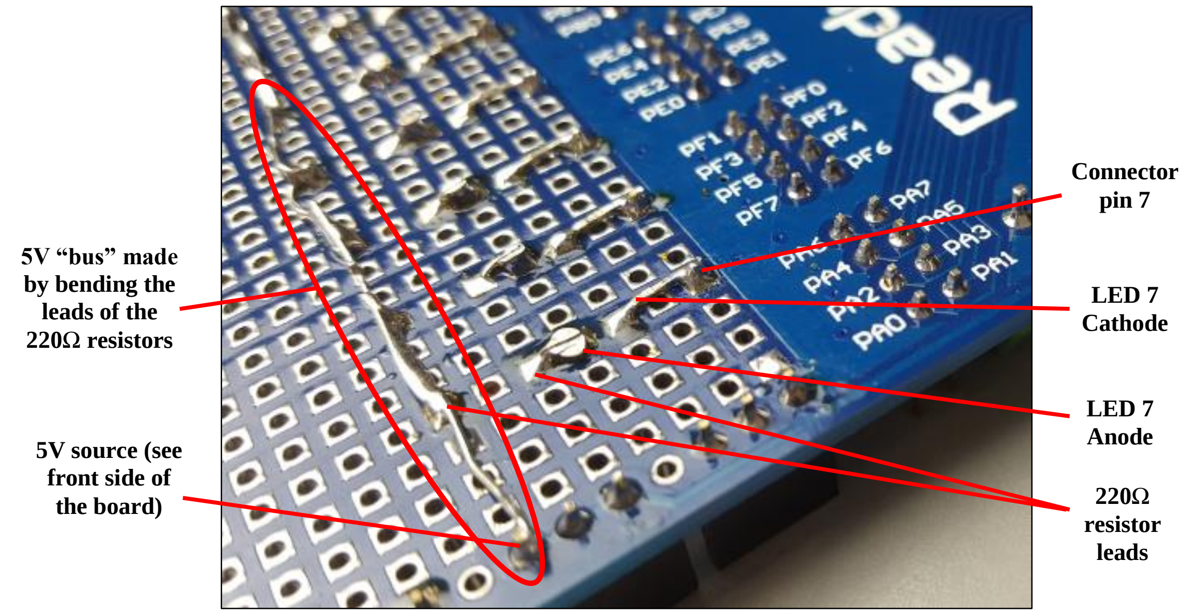
5V "bus" made by bending the leads of the 220Ω resistors
5V source (see front side of the board)
Connector pin 7
LED 7 Cathode
LED 7 Anode
220Ω resistor leads

Figure 2. The backside of the ReadyAVR board showing how the 5V power source, 220Ω current-limiting resistors, LEDs, and connector pins are attached.

Software Architecture:

You'll be using the ReadyAVR **joystick** again, but instead of controlling the blink rate of an LED, you'll use the joystick's **center switch** to change between four (or more!) predefined flashing patterns for the eight LEDs externally connected to **PORTE** of the ATmega128A.

The patterns are:

0. Low to high (E0 to E7, with only one LED on at a time).
1. High to low (E7 to E0, with only one LED on at a time).
2. Back and forth (E0 to E7 to E0, with only one LED on at a time). Do **not** let the LEDs on each end stay on for double the amount of time!
3. Your choice. Make it interesting! Your instructor will show you some examples. Do more than one for some extra credit—see the Report section below for more info.

Important Code Design Information

The code to actually flash the LEDs must be in your `main()` routine, and an ISR, triggered by pressing the joystick button, should change a **state variable** that `main()` uses to decide which pattern to display. This design pattern is called the *reactor pattern*, and should be familiar to you from your programming classes. A second ISR must be used to keep track of time in 1mS intervals or "Ticks." Use this "Tick" counter to **ensure that LEDs stay on for 50mS**. Study Figure 3 for more details on how this should be implemented.

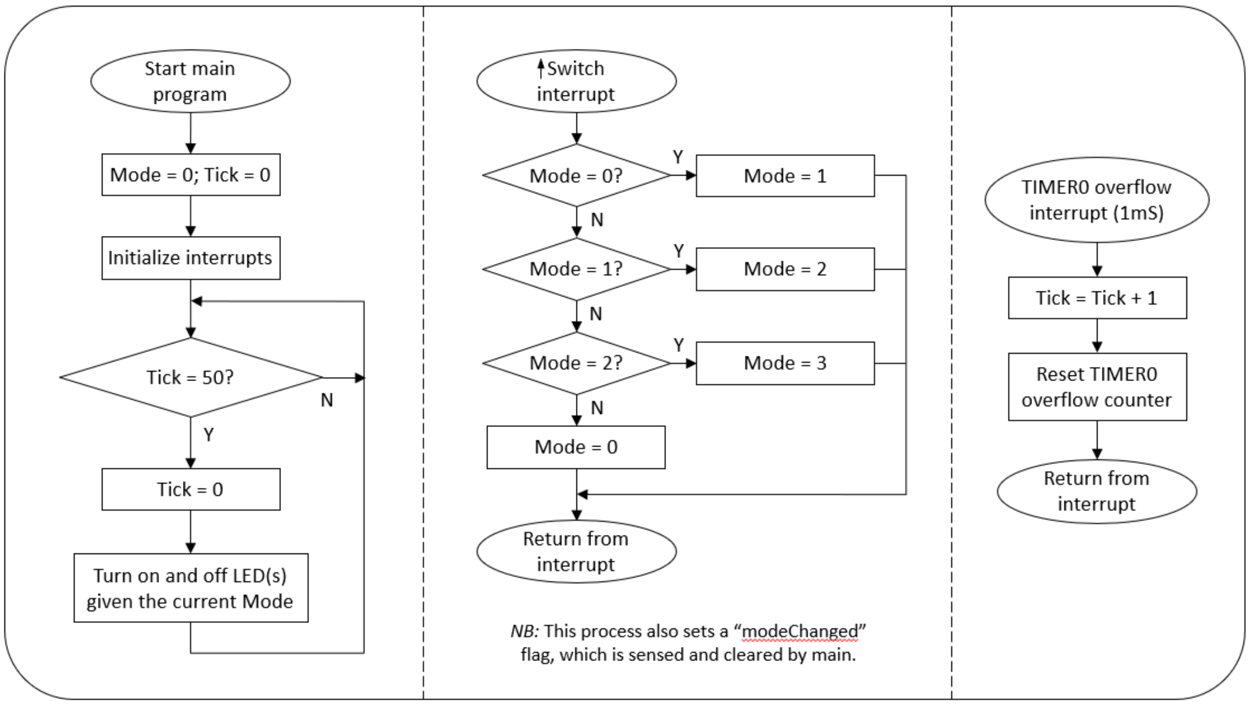

Figure 3. StateChart representation of the program's logic showing three concurrent processes.

Lab Procedure:

Start by accepting the GitHub Classroom assignment you received via email. Note that for this assignment the "starter repo" will only contain a README.md file and this lab handout in PDF format! (Don't worry—you'll soon have some code to fill it up with!)

On your computer, open your **EECE3624Labs** solution in Atmel Studio. Click **File | Add | New Project**, select **C/C++** from the list of **Installed Templates** and choose **GCC C Executable Project**. Name the project **Lab06** and click **OK**. When asked for the processor type, select **ATmega128A** and click **OK** again. Atmel Studio will create and add a new C project to your solution, and will open the `Lab06.c` file in the editor. Choose "Save All" in Atmel Studio to save Lab06 project.

Open Git Shell, navigate to the **EECE3624Labs\Lab06** solution directory that was just created and use the proper git commands (below) to create a local Lab06 repo, link the local repo with your remote Lab06 repo, and populate both with the newly created Lab06 project files.

```
>> git init
>> git remote add origin <your-Lab06-URL>
>> git pull
>> git checkout main
>> git add .
>> git commit -m "Lab06 project init"
>> git push
```

As always, if something doesn't go right, ask for help. And remember, periodically use the `git commit` and `git push` command in GitShell to push your latest changes to GitHub!

Now that the project files exist, the local repo has been initialized, and the local- and remote repos are in sync, you're ready to configure the ReadyAVR board for this lab. Using eight female-to-female jumpers, connect the eight LEDs on the ReadyAVR board to the eight pins on PORTE. **Be sure PortE.0 is connected to LED0** on the ReadyAVR board (see Fig. 1), and **PortE.7 is connected to LED07**. Then develop your code as follows:

1. Create an ISR to set a global state variable named "Mode" to 0, 1, 2, or 3 as you continue to press and release the center joystick button. **Only change Mode when the button is <u>released</u>**, and after 3 go back to 0. *NB:* The center joystick button is wired to a pin that cannot generate an interrupt itself, so **externally connect the port pin associated with the center joystick switch to the PORTD0 pin** and use interrupt INT0 to respond to release of the joystick's center pushbutton.
2. Create a second ISR that fires **<u>once per millisecond</u>**. Use the **TIMER0 overflow** condition to trigger the interrupt, and inside this second ISR increment a global variable named "Tick." [*Microprocessor Geek Note:* Your TIMER0 interrupt handler will need to reload the TCNT0 register with its initial count value so that TIMER0 will overflow again in exactly 1mS. **Very important!**]
3. Once your interrupt routines are working, write your `main()` program to implement the LED flashing patterns mentioned earlier. Your `main()` code should configure the interrupt registers and I/O lines, then enter an infinite loop (often called a *"game loop"*) that uses Mode and Tick to decide which pattern to use and which LED(s) to turn on and off every 50mS.

   Figure 3 shows this logic. The "little" loop in `main()` spins until "Tick" reaches 50. Once 50 is reached, `main()` resets Tick to zero, determines which pattern (via "Mode") is being used, and then turns off the currently lit LED and turns on the next LED in the pattern. **<u>The larger loop in `main()`is the game loop!</u>** If this isn't 100% clear, review Figure 3. Still not clear? Ask!

Report:

This is a **one-week lab**. You must **demonstrate your working code** to your instructor and **submit your lab report via Canvas and your code via GitHub** on or before the start of lab on **Friday, October 29**. In your report, include a concise description of how you set up and used TIMER0, and include your <u>properly commented</u> code in an **Appendix**. As always, you'll be graded on the correctness, completeness, readability and organization of both your report and your code, so make it **read well** and **look good**!

And if you do more than one of your own patterns, **you can earn up to a full letter grade in bonus points**, depending on how many extra patterns you develop, how interesting they are, how well they function, and how well you describe the design of each pattern in your lab report.

Have fun and blink on!