CS1 Lab 4: Timing with the Arduino

In the previous lab, we learned how to do simple branching on the Arduino along with calculating a simple mathematical function. In this lab, you will explore timing on the Arduino, and how to make the LEDs blink specific patterns.

1 Getting Started

Make sure you have everything you need to complete this lab. These items should be included in your CS1 Lab Kit. You will only need to wire up 1 LED for this week's lab:

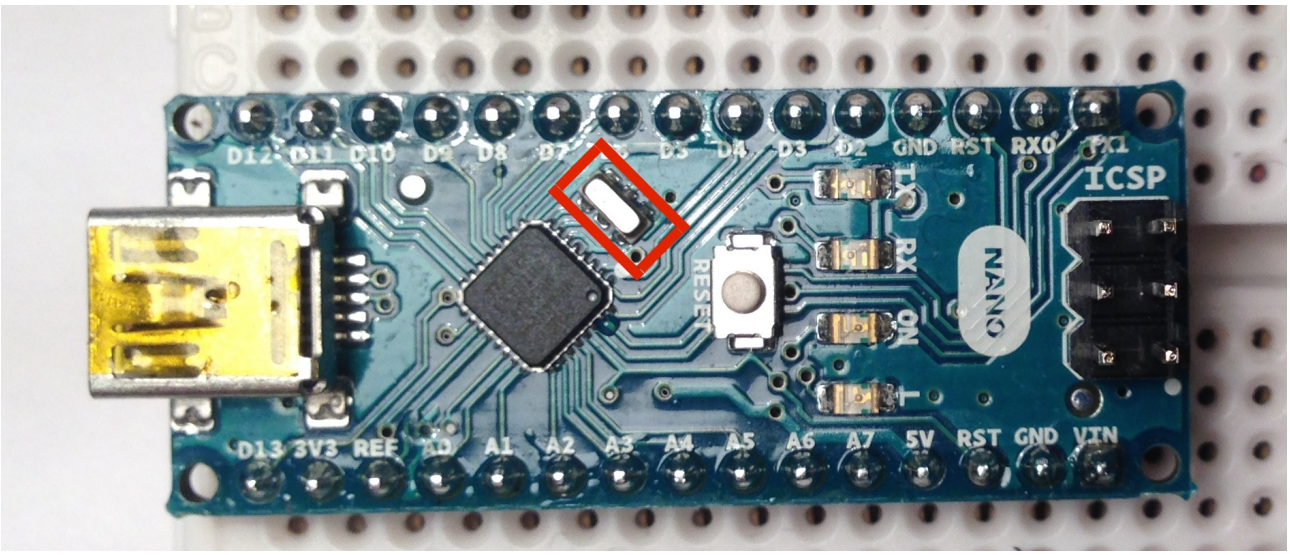
- Arduino Nano.
- USB cable.
- Atmel ATmega328 Datasheet.
- Atmel 8-bit AVR Instruction Set Manual.
- Arduino Schematic Circuit Diagram.
- Breadboard.
- 1 LEDs.
- 1 330Ω resistors (color coded: Orange-Orange-Black-Black-Brown).
- 2 jumper wires.

2 Pre-Lab: Calculating Timings with the AVR chip

In this lab we want to have our Arduino blink the LEDs at specific timed intervals. In order for us to do this we need to understand a little bit about how fast the AVR chip processes instructions through its *clock rate*. A CPU's clock speed rate measures how many clock cycles a CPU can perform per second. It is measured in *Hertz*, which a unit for clock cycles per second.

Pre-Lab Question 1 First take a look at the Data Sheet for the AVR, what is the *maximum* clock rate that the AVR can operate at?

A CPU can either be clocked internally or externally. In the Arduino Nano's case, it's clock rate is set by an *external crystal* electronic component. You can see the crystal component on the Arduino as a small silver rectangle close to the AVR chip. Take a look on your Arduino Nano, and find the crystal using the photo below where it is highlighted in red:



We need to find out what the actual clock rate that the AVR is operating at by looking at what frequency (in *Hertz*) the crystal oscillates at. Look at the Arduino *Circuit Schematic*. You will find the symbol for the crystal connected to the two `XTAL` pins on the AVR chip.

Pre-Lab Question 2 Based on the *Circuit Schematic*, what is the actual clock rate that the Arduino is operating at?

Pre-Lab Question 3 Based on this clock rate, how many clock cycles per second is the Arduino operating at?

Remember that the AVR chip is a **RISC** based architecture. That means that since it has a reduced instruction set, *most* of the machine instructions operate within 1 clock cycle. You can find out how many clock cycles each AVR instruction takes by looking at the *Instruction Set* documentation, in the **Complete Instruction Set Summary** table starting on page 11.

Pre-Lab Question 4 Look up the *No Operation* (`NOP`) instruction, in how many clock cycles does it require to operate?

Pre-Lab Question 5 Based on clock cycles required for the `NOP` instruction, how many seconds is one `NOP` instruction in duration?

You now have an idea at how much time a single Arduino instruction requires to execute. Using this information, we can look at how to delay the Arduino to make the LED blink.

3 Pre-Lab: Making the LED Blink

Assume your Arduino is set-up with LEDs wired up to `PORTB` in the set-up from Lab 2. We would like to make one of the LEDs blink where it is **ON** for one half a second and then **OFF** for another half a second. Take a look at the assembly code snippet below:

```
1         ldi r16, 0x01
2         out PORTB, r16
3
4         ; delay for 0.5 second
5
6         ldi r16, 0x00
7         out PORTB, r16
8
```

```
9           ; delay for 0.5 second
```

Listing 1: Assembly LED Blink

A value is written out to `PORTB` from register r16 to turn an LED **ON** and then **OFF**.

Pre-Lab Question 6 First, based on the above code, which LED connected to the Ardruino will be turning **ON** and then **OFF**?

How will we make the Arduino delay for 0.5 seconds? We can make the Arduino wait and do nothing by using the `NOP`, *No Operation* instruction.

Pre-Lab Question 7 Based on the timings we calculated in the previous section, how many `NOP` instructions would we have to insert at Line 4 in the code above to delay the Arduino for 0.5 seconds?

Do you think it is feasible to write in that many `NOP` instructions in an assembly program? What is a programming structure that you have seen before (perhaps in Java) that will allow us to run the Arduino for that many clock cycles in order to delay 0.5 seconds without having to write a whole ton of NOP instructions?

4 Pre-Lab: Writing a Loop in the Arduino

Consider the following Assembly code, which performs an iterative loop in assembly:

```
1               ldi r17, 100
2       loop:   nop
3               dec r17
4               cpi r17, 0
5               brne loop
```

Listing 2: Iterative Loop in Assembly

Pre-Lab Question 8 Using the *Instruction Set* documentation, how many times will the `NOP` instruction in Line 2 (at the label `loop`) execute?

Now that you have an idea how many times the above loop iterates, let's estimate how long in clock cycles total the is in duration.

Using the *Instruction Set* documentation, calculate how long in clocks cycles Lines 2-5 are for a single iteration through the loop. You may have to look at each assembly command's documentation page in the *Instruction Set* documentation. Assume for this calculation that the branch *condition* in Line 5 is *true* (i.e. the program flow branches back to the `loop` label).

Pre-Lab Question 9 How many *clock cycles* is a single iteration through Lines 2-5 in the above loop?

Using this estimate of the single iteration in clock cycles, you can now calculate the total duration in seconds for the loop in Listing 2.

Pre-Lab Question 10 What is the total duration (in *seconds*) for the complete execution of the above loop in Listing 2?

You'll find that this duration is no where close to our desired delay of 0.5 seconds.

Pre-Lab Question 11 How many *iterations* through the above loop, would be enough to get a delay of about 0.5 seconds?

Think about how we would change the above loop to increase to these number of iterations. A simple answer to this is to increase the constant loaded in `r17` in Line 1 above. However, one thing to consider is how large a value the Arduino registers are. Think about how many bits in size register r17 is in size.

Pre-Lab Question 12 By increasing the constant loaded in `r17`, what is the maximum number of iterations we can have the loop in Listing 2 perform?

How would you modify the code in Listing 2 to delay for 0.5 seconds? Try to start an assembly program that performs this delay before you come to lab. In the first part of this week's Lab you will be working on an assembly program to make the LED blink ON and OFF with this 0.5 second delay.

Use the answers from these **Pre-Lab Questions** to complete the Pre-Lab 4 Quiz on KEATs. These Pre-Lab Questions will aid you in the completion of the lab in your lab session.

5 Lab: Program the Arduino to Blink ON and OFF for 0.5 seconds

1. Using a copy of a previous program from Lab 2, create a new file and name it `blink.s`. Your program should do all of the set-up that is required to write to `PORTB`, in particular:
   (a) Declare the appropriate code symbols with `.eq`
   (b) Clear the `SREG` register at the beginning of the program.
   (c) Set-up `PORTB` for **output** with the `DDRB` register
2. Start with the code from the Pre-Lab's Listing 1 to turn ON and OFF the LED attached to `PORTB`'s `PB0` pin.
3. You will have to implement 2 separate loops that delay for 0.5 seconds each. One delay will be after the LED is turned ON, and one will be after the LED is turned OFF. Start with the code from Listing 2, and try to construct enough iterations to delay for 0.5 seconds. Remember that you can use more registers in your code to keep track of values (You can use any of the registers r16 through r31).
4. Test your code iteratively. As a TA if you need any help.
5. Once you have the LED blinking ON and OFF once, modify your code to have the LED blinking constantly. If your timing is correct, you should have the LED turn ON every second, and you can see how your timing compares to a watch or a clock's second hand.

Congratulations! You have successfully written a blinking LED program! Make sure to demonstrate your code to a TA for any feedback.

6 Use a Sub-Routine

You will notice that your `blink.s` program has two blocks of code that do the same thing: delay for half a second. We will start to modularise our code by implementing a *sub-routine* to delay for 0.5 seconds. This way, the code in the main loop of our program can call the sub-routine every time we desire a delay for half a second.

1. Copy your program from the previous section can call the new file `blink_halfsec.s`
2. Look up in the *Instruction Set* documentation, the commands `CALL` and `RET`. In particular, look at the *Example* code section on how to use the commands.
3. Use these commands to implement a sub-routine that is stored at the code label `halfsec`. Have this sub-routine perform the 0.5 delay you programmed in the previous section.
4. Use your `halfsec` sub-routine in your main loop to delay the Arduino after turning the LED ON and also after turning the LED OFF.

Congratulations! You have successfully implemented a sub-routine in Assembly. Now your code is a little bit more organised and readable.

7 Create a 10 Millisecond Delay Sub-Routine

You can make your code more re-usable by adding a parameter to your sub-routine that you worked on in the previous section so that you can set a variable delay. A good level of resolution for a delay is at the *10 millisecond* time interval. Using this parameter will allow you to delay various lengths at 10 ms steps (i.e. 100 ms, or 500 ms).

At the assembly code level, you have to decide how you will parameterise your sub-routines. A very simple way to do this is to decide on a register that you will set with the parameter value. Then, when you call your sub-routine, it will use this register in its code (think of registers as global variables).

A more dynamic and complex way of parameterising sub-routines (and creating proper functions) is to use a *stack* (implemented in hardware or in program memory). For this lab, we will use the simple way of adding a parameter to your sub-routine delay.

1. Copy your program from the previous section and call the new file `blink_delay.s`
2. Rename your half-second sub-routine so that is stored at the code label `delay`.
3. Modify the loops in your sub-routine so that it can delay for 10 ms (i.e. 10 milliseconds).
4. Decide on a register that your `delay` sub-routine will use as a *10 ms* parameter. Write you sub-routine so that it uses this register as the number of 10 milliseconds delays the sub-routines will iterate over.
5. Modify your main loop so that the LED is ON for **400 ms** and the LED is OFF for **200 ms**.
6. Test your code, now your LED should blink more like a heart-beat.

Congratulations! You have successfully parameterised a sub-routine in Assembly.

8 Lab Assembly Program: My Morse Code

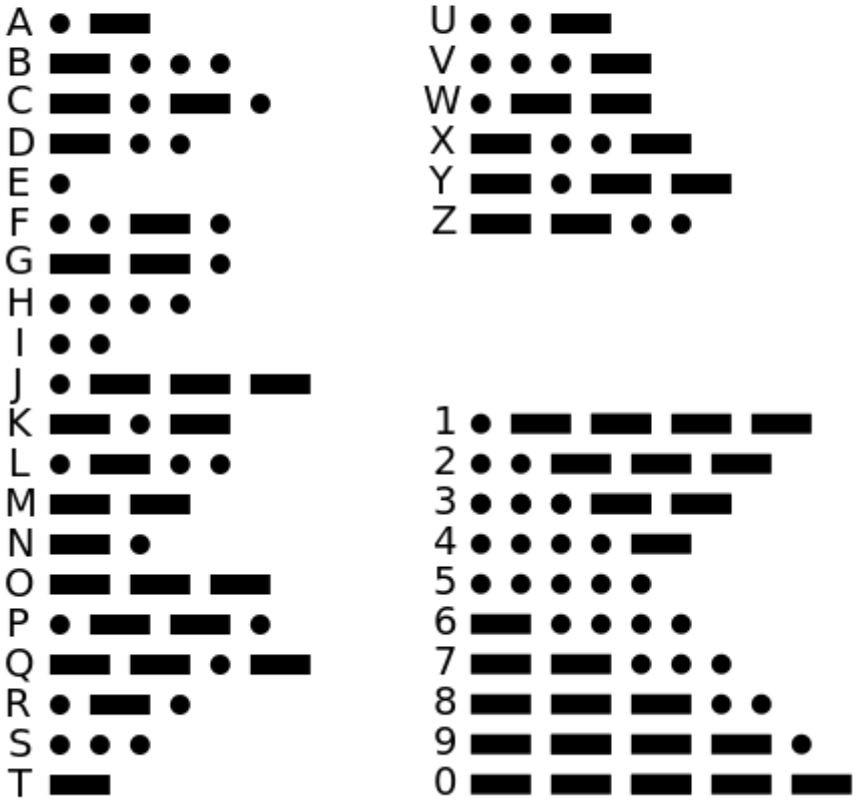
For the Lab 4 Assembly program that you will turn in, you will use the programs that you wrote in the lab to broadcast a short message via Morse code on the LED.

What is Morse code? Morse code is a method of transmitting text information as a series of on-off tones, lights, or clicks that can be directly understood by a skilled listener or observer without special equipment. (`https://en.wikipedia.org/wiki/Morse_code`).

Below is the International Morse Code Roman alphabet:

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.



Your program will blink a 3 letter sequence in Morse code on the LED. To find out what Morse code letters you will program, on KEATs under Lab 4 you will find a link *What is my Morse Code?*. *Make sure that you are logged into KEATs with your account*, and click on this link. You will find a 3 letter sequence that you will write a program to transmit the Morse code sequence for.

So that we can perceive the Morse code, we will use a unit length of **200 milliseconds (ms)**. This means the duration of a *dot* is **200 ms**, and a *dash* is **600 ms**.

Make sure that:

1. Your program is called `my_code.s`.
2. Your program writes the Morse code sequence to the `PB0` pin of `PORTB`. This is the same set-up as in the other programs you wrote in this lab.
3. Your program will continually loop through the 3 letter sequences.
4. Assume that there is **no** word break after your 3 letters, just repeat the 3-letter sequence.

For example, if your sequence was `ABC`, then your program would run as follows:

1. Turn ON the LED for **200 ms** for the first *dot* of the letter `A`

2. Turn OFF the LED for **200 ms** for the inter-part space of the letter `A`
3. Turn ON the LED for **600 ms** for the first *dash* of the letter `A`
4. Turn OFF the LED for **600 ms** for the inter-letter space between the letters `A` and `B`
5. Turn ON the LED for **600 ms** for the first *dash* of the letter `B`
6. ... and so forth
7. Until the last *dot* of letter `C`
8. Turn OFF the LED for **600 ms** for the final inter-letter space.
9. *Loop back to the beginning of the Morse code sequence*

9 What to turn in

When you have tested your program completely, **ONLY** turn in the `my_code.s` assembly program on KEATs through the Lab 4 Assembly Program submission link.

DO NOT put your program in a .zip or .7z or a Word document. Doing so will result in losing marks for your Lab 4 Assembly Program submission.

10 CHALLENGE: LED Ping Pong

Only attempt this challenge if you have completed the entire Lab, and turned in your Lab 4 Assembly Program. This challenge is worth **0** marks, but it gives you a chance to show off your assembly programming skills!

Now that you know how to delay the timing, can you write a program that turns on the LEDs attached to `PORTB` in a pattern where it appears that a single LED light is chasing back and forth?