0. General Information

Student Name:	Ellis Hobby
Student ID Number:	01928869
Team Name:	Tube Disaster
Team Member Names:	Ellis Hobby
Date of Completion:	02/22/22
Demonstration Method:	Zoom

1. Design

1.1 Hardware Design

The hardware design incorporates several individual circuit blocks to achieve full system functionality. A drawing of the complete schematic can be found in the appendix.

Figure 1 shows the ATmega2560 microcontroller (MCU) portion of the schematic. The ATmega2560 is a 16 MHz 8-bit microcontroller with a plethora of useful features. Most notably for this project 86 general purpose (I/O) pins, two 8-bit timers, four 16-bit timers, serial peripheral interface (SPI), and external interrupt functionality [1]. Net labels are used to indicate where each pin connects to the external circuits. This was used to improve readability of the schematic.

Figure 2 shows connections for the six light emitting diodes (LEDs) used as the traffic light indicators. These were placed *on PORTF 0:5* (Arduino names A0-A5) and given net labels *RED1, YLW1, GRN1, RED2, YLW2, and GRN2*. Having The LEDs grouped physically close and on the same PORT was thought to help improve ease of hookup and programming. Each LED has a 1 kΩ current limiting resistor from its anode to its corresponding pin on the MCU and its cathode tied to ground. This means an LED will light up when the corresponding pin on the MCU is set to a high state.

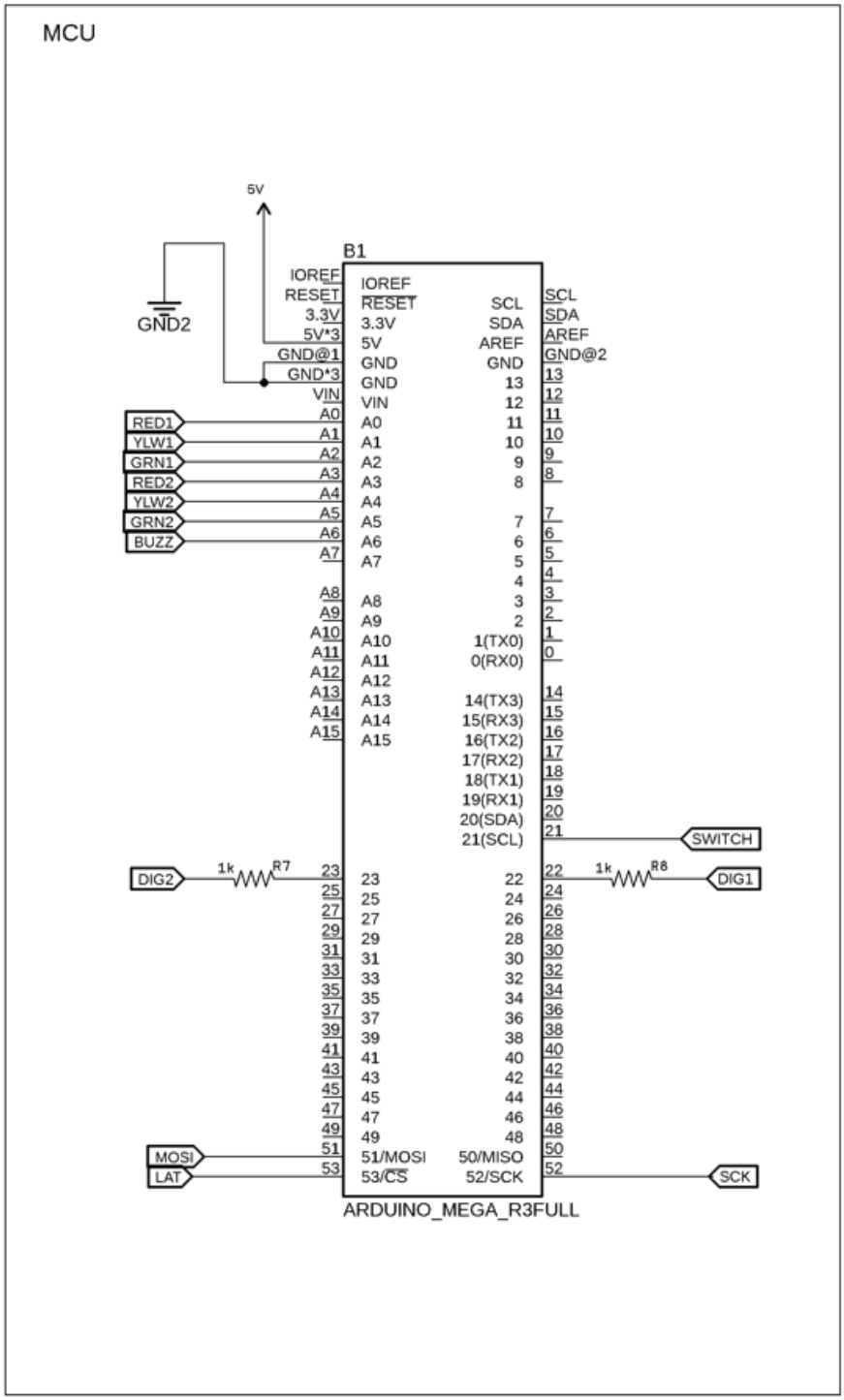


Figure 1: ATmega2560 Traffic Controller

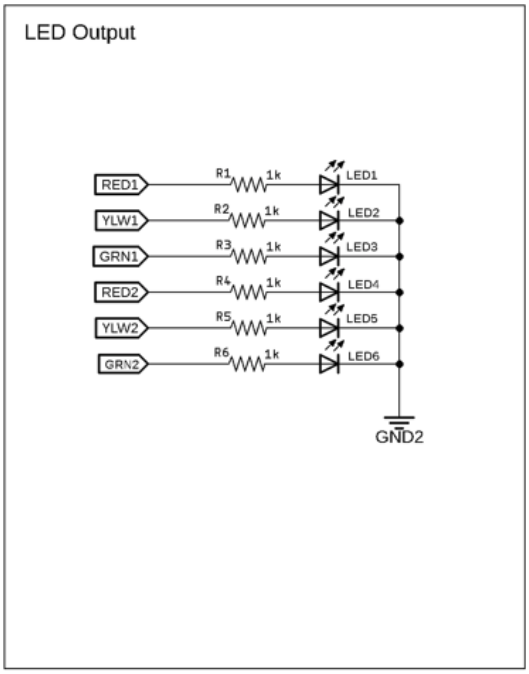


Figure 2: Traffic Light LEDs

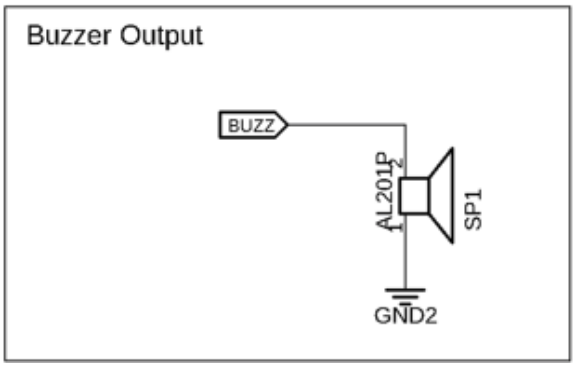


Figure 3: Traffic Buzzer Output

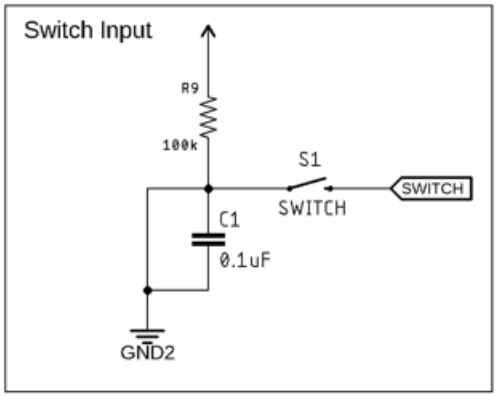


Figure 4: Start Button Input

The active buzzer circuitry is depicted in Fig. 3. This component was placed with its positive control terminal on *PORTF 6* (Arduino name A6), and its opposite terminal connected to ground. This enabled the buzzer tone to be signaled by setting the relevant pin on the MCU high.

The start button circuit, shown in Fig. 4, was placed on *PORTD 0* (Arduino name 21). This pin was intentionally chosen due to it being able to function as an External Interrupt (*INT0*). The nature of how this was utilized within software will be discussed below in the section 1.2 Software Design. An additional resistor and capacitor were used to add hardware debouncing to the switch as a means to improve reliability and prevent electrical chatter. These components act as a lowpass filter to help smooth out the signal. As shown by Equation 1 the 0.1 μF capacitor and 100 kΩ resistor create a time constant of approximately 10 ms to add a slight delay that is not directly perceivable to the end user [2].

$$\tau = R * C \tag{1}$$

The four digit 7-segment display counter, Fig. 5, was implemented using a 74HC595 shift register. This enabled full control of the display with minimal designated pins on the MCU. Three pins were reserved for the shift register data, clock, and latch on *PORTB 0:2* (Arduino names 51-53). Two additional pins were reserved for the first two digit selectors (DIG1, DIG2) on *PORTA 0:1* (Arduino names 22-23). These were connected through 1 kΩ current limiting resistors. Through testing it was found that the display was common cathode, so the anodes for each segment (*A*-*G*) on the display were connected to the data outputs (*QA*-*QG*) on the shift register to enable proper control. The decimal point (*DP*) on the display was not used, so its pin was simply tied to ground. The anodes for each digit are connected in parallel as seen in Fig. 6 [3], this meant the digital control pins (DIG1, DIG2) needed to be rapidly strobed in software to create the illusion of displaying two separate digits. The unused digital control pins (DIG3, DIG4) where tied to ground as they were unused and will remain off.

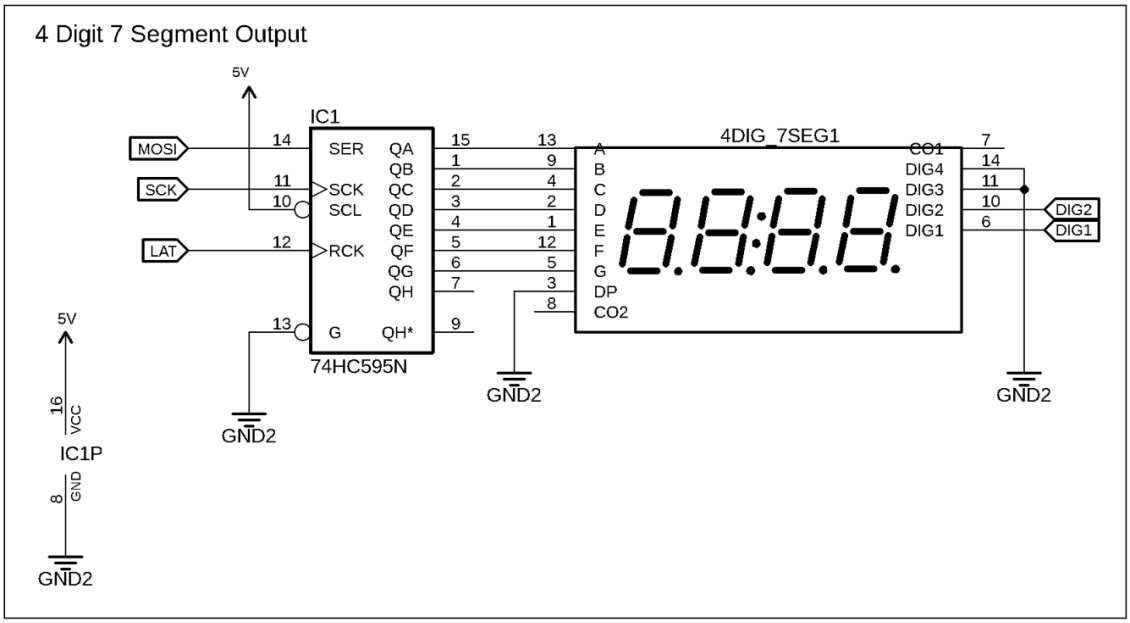


Figure 5: Four Digit 7-Segment Display Output

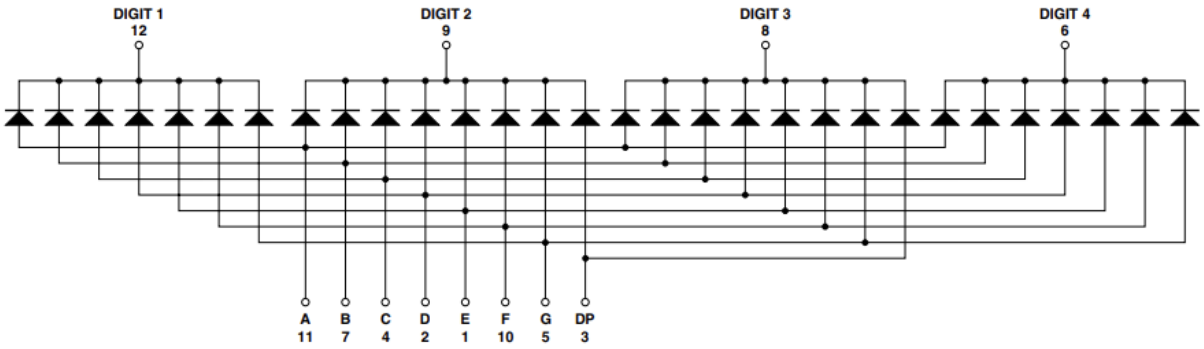


Figure 6: Common Cathode Four Digit 7-Segment Display Internal Circuit

1.3 Software Design

The traffic light controller can be best organized using a finite state machine (FSM). Figure 7 shows the FSM created to help clarify system operation before any actual programming began. The diagram was drafted using the Lucidchart application (https://www.lucidchart.com). Here we can see the main program flow exists in six discrete states: *Clear, Flash, Grn_Red, Ylw_Red, Red_Grn, and Red_Ylw*. The FSM also account for two asynchronous events that effect the main program flow; *Timer Interrupt* and *Button Press Interrupt*. Three variables are used to trigger and dispatch system events: *count*, *tseg*, and *buzzOn*. One input is used, button, which is controlled by the *Button Press Interrupt*. The six outputs correspond to the LEDs and are called according to the relevant states.

On initialization the *count*, *tseg*, and *buzzOn* variables are set to 0 and the *Clear* state is entered. The *count* variable simply counts the time passed and is incremented by the *Timer Interrupt* once per second. The *tseg* variable is used as a countdown to help display the time remaining until a light switches on the 7-segment display. It is subtracted from the running count once second within the *Timer Interrupt*. The *buzzOn* variable is a Boolean that gets checked each loop in order to indicate when the buzzer should be triggered.

The system loops between the *Clear* and *Flash* state indefinitely until the button has been pressed. While in the initial loop both red LEDs will flash once per second. Once the button has been pressed the system enters the main traffic loop and will not return to the initial *Flash-Clear* cycle. In the main traffic loop a light will be in the “go” state for 23 seconds. Where it will stay green for 20 seconds, and then turn yellow while signaling the buzzer for 3 seconds. The opposite light will stay red, the “stop” state, for the full time. The 23 second cycle is visually displayed by the four digit 7-segment counter, which counts down from 23 to 1 and repeats. Each side of “cross traffic” goes through this 23 second cycle and repeats indefinitely until the system is powered off.

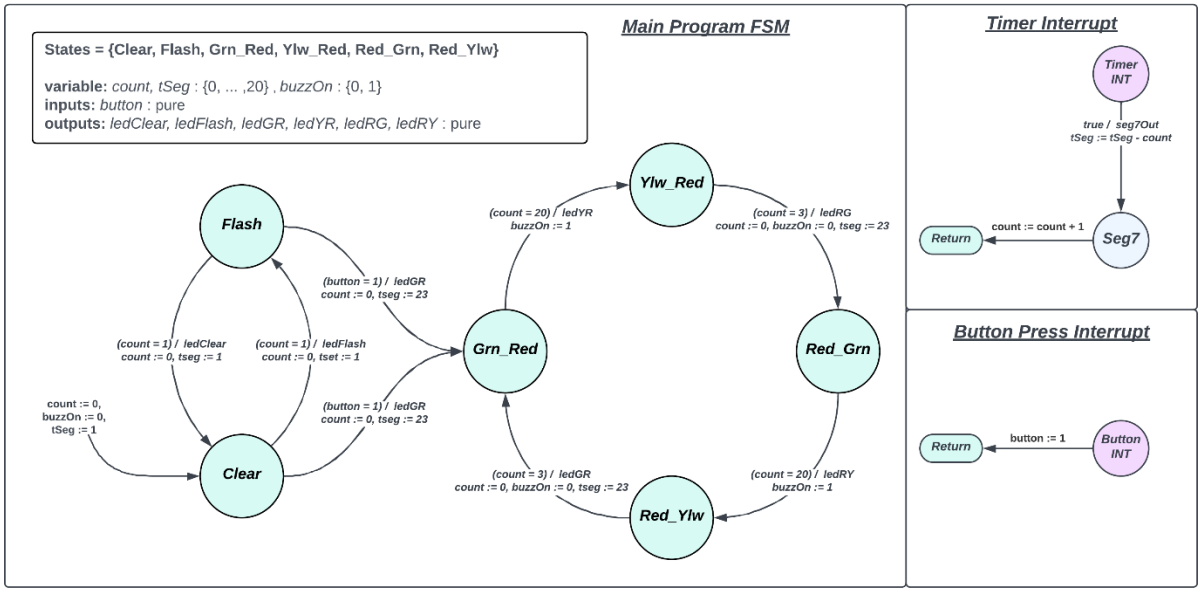


Figure 7: Traffic Controller FSM Diagram

The FSM code made use of two enums for States and Events, and a struct that held relevant variables for transitions and outputs. A simple "switch-case" was setup to check the current state, compare it to the desired event, and dispatch accordingly. It is worth noting that initially a building a class inheritance type system was considered but this seemed like overkill given the small number of states.

Some key notes for the hardware side of the program code were the use of low-level PORT manipulation. It was thought that this would be a good learning exercise while simultaneously being more efficient as it does not deal with the overhead of Arduinos function calls such as *digitalWrite(..)*. Furthermore, achieve something such as setting the LED colors can be achieved using less lines of code if they are organized onto the same port. For example, to turn all six LEDs on, one would have to call *digitalWrite(..)* six times rather than simply calling *PORTx = [LED pins]*.

Recognizing the button input was achieved through the use of an External Interrupt (*INT0*). This was thought to be more effective as it would use less overhead compared to polling for a

button press each loop. Additionally, within the interrupt service routine (ISR) the relevant External Interrupt (*INT0*) could be disabled and would no longer have to be worried about.

For communication with the shift register SPI was implemented this was thought to be faster and use less overhead than calling *shitOut(..)* or manually bit bashing the data. To enable the SPI transactions to transmit the appropriate bits for each digit and synchronize strobing of the digit control pins, bit shifting was used to pull the relevant 8-bits from a 16-bit number within a function call for the SPI output.

The overall project was split into a few different files to assist in program organization; Lab1_code.ino, fsm.h, mcu.h, mcu.cpp, and seg7.h. The core of the program sits in Lab1_code.ino, this is where initialization calls are made and the switch-case loop is checked. The enums and struct for the FSM are kept in fsm.h. Anything pertaining to MCU peripherals such as I/O, SPI, timer, and interrupts were defined in mcu.h. Here defines are used to help clarify PORT manipulation calls. For example, *PORTF* and *DDRF* were defined as *LIGHT_PORT* and *LIGHT_DDR* for the LEDs. Any functions pertaining to the MCU peripherals, such as *set_leds (uint8_t color)*, were kept within the mcu.cpp file. The seg7.h simply defines the relevant hex values for control of the segments of the 7-segement display and uses them to create a const array named *Seg_Count* for indexing the 23 second countdown.

1.3 Results

Figure 8 shows the prototyped circuit built on a breadboard; colored jumper wires were used as a means to clarify signal nets for troubleshooting purposes. Circuit blocks where partitioned to certain areas to assist in further build organization, Fig. 9. The relevant code for testing each circuit block was tested individually before implement the system as a whole. For instance, the initialization and switching of LED PORT color values was tested on and revised on its own. This enabled bugs pertaining to an individual circuit block to be caught and remedied before complexity was increased. After verification the separate blocks were combined into the

system whole and fully tested. The whole traffic controller system was found to function exactly as anticipated and needed only minor tweaks to perform at the required specification.

Improvements could be made if this system were to be actually implemented under the same specifications. The biggest and most obvious would be using a smaller MCU device. The ATMega2560 is a bit overkill for this project given the minimal use of digital I/O and peripherals. Suitable replacements would be the ATmega32u4 [4] or ATmega328 [5] which cost $5.05 and $2.26 respectively for a TQFP package. These come in at over half the cost when compared to the ATmega2560 TQFP package at $18.86 [6].

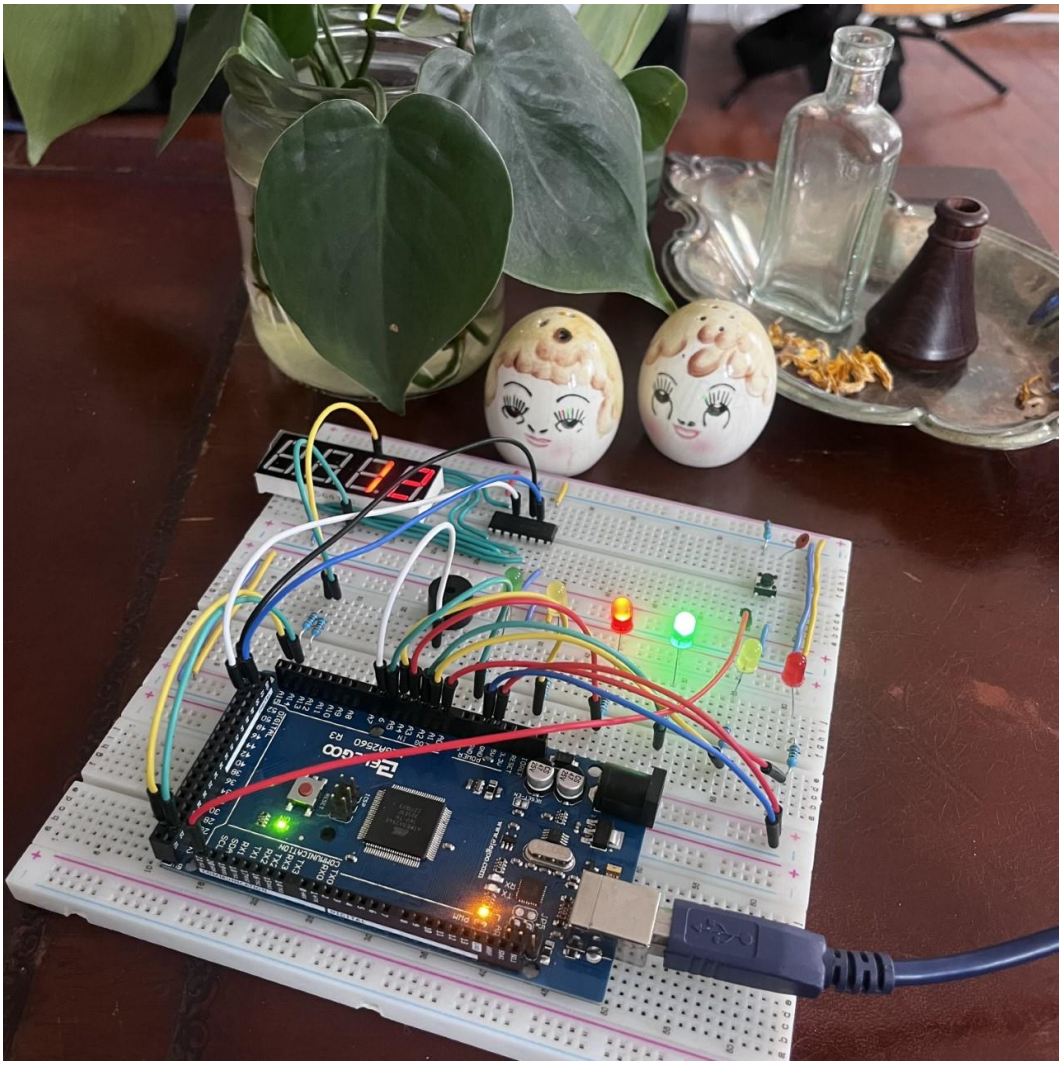


Figure 8: Traffic Controller Breadboard Prototype

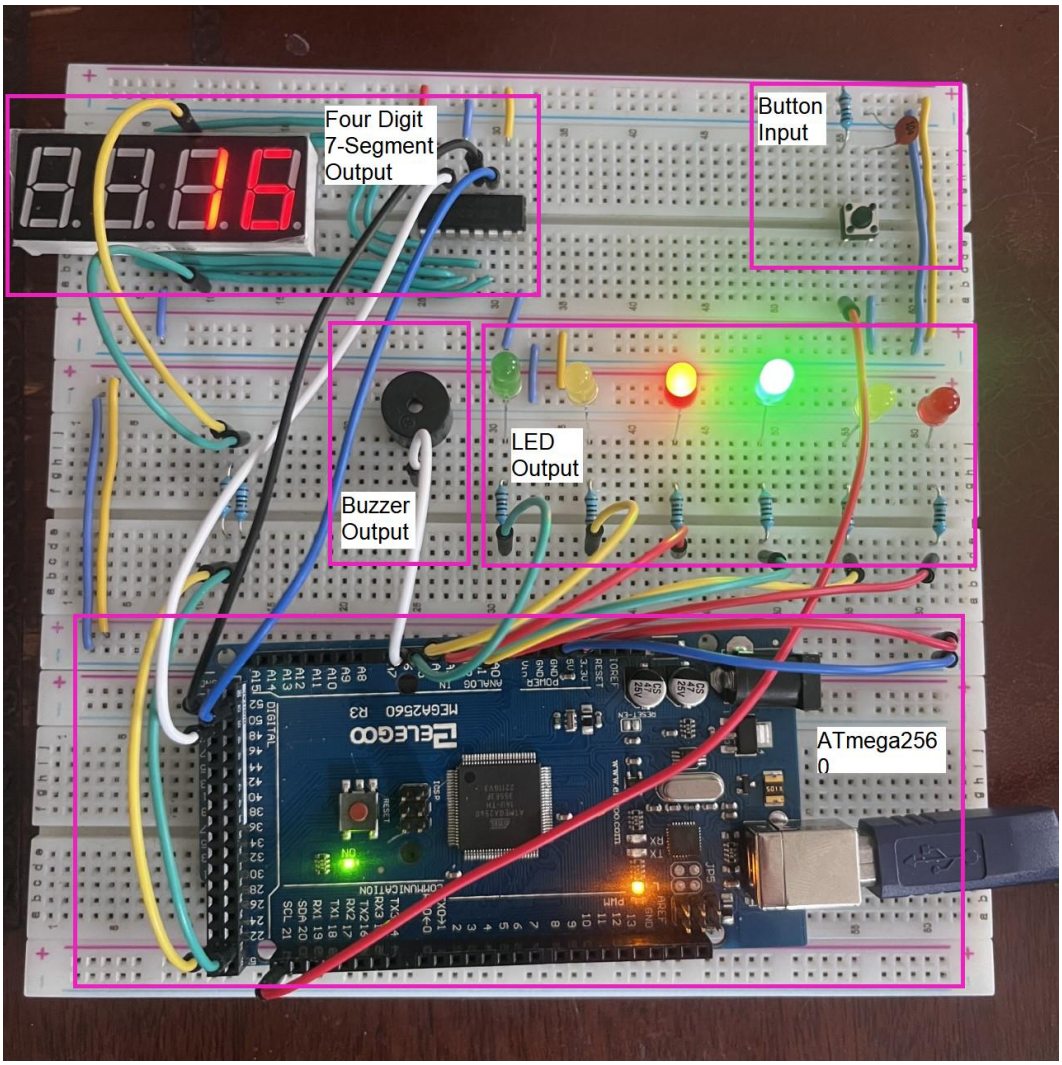


Figure 9: Traffic Controller Breadboard Circuit Blocks

2. Problems Encountered and Solved:

On the initial build and test phase it was noticed that there was some cross chatter between each digit on the on the 7-segment display, specifically if digit one was displaying the number one the relevant segments would be “ghosted” on segment two. It was considered that this was due to the strobing frequency for each digit being too slow. Technically both digits will display the same value at a moment in time but if they are strobed fast enough the illusion of separate display values is created. Reviewing the code revealed where this could be remedied within the *spi_7seg_set(uint16_t value, uint8_t set_digit_pin, uint8_t digit)* function call. Initially a slight

delay (100 ms) was implemented after setting the shift register *LAT* (RCK) low and before setting it high between SPI transmission. It was thought that this would help with shift register synchronization and prevent potential data loss. However, removing these two delays and preforming multiple tests alleviated the cross chatter and did not appear to cause the system to suffer from data loss.

Another small issue found with the 7-segment display values pertained to the actual numbers being displayed. At first, they did not appear to be synched up with the actual time. Furthermore, the first number at the start of a countdown (23) was displayed as "gibberish". Reviewing the code revealed an indexing error for the *Seg_Count* array. If the *tseg* variable is set to 23 at the start of a countdown and this caused the array to be indexed out of bounds. Because the *Seg_Count* array has 23 members (0-23 Hex values for 7-segment) the max index value is 22. The simple fix for this was simply subtracting one from *tseg* when indexing (*Seg_Count[tseg – 1]*).

3. Personal Contribution to the Lab (Technical Details):

The graduate section of this class (EECE.5520) allows students to work solo on lab projects. This was the case for this project, therefore all contributions discussed above were contributed by the author Ellis Hobby. Full schematic files and project code can be found under the *Lab1* file tree in the github repository at: https://github.com/EllisHobby/Microprocessors-Labs-2023. This repository is currently private, therefore access will need to be requested to review the files. To request access to the repository please email the author at: ellis_hobby@student.uml.edu.

4. Lessons Learnt:

Operational features and limitations of the ATmega2560 were investigated. Reviewing the datasheet reveals how powerful this MCU actually is. The sheer number of peripherals is where this particular piece of hardware shines, for example offering 86 general purpose digital I/O.

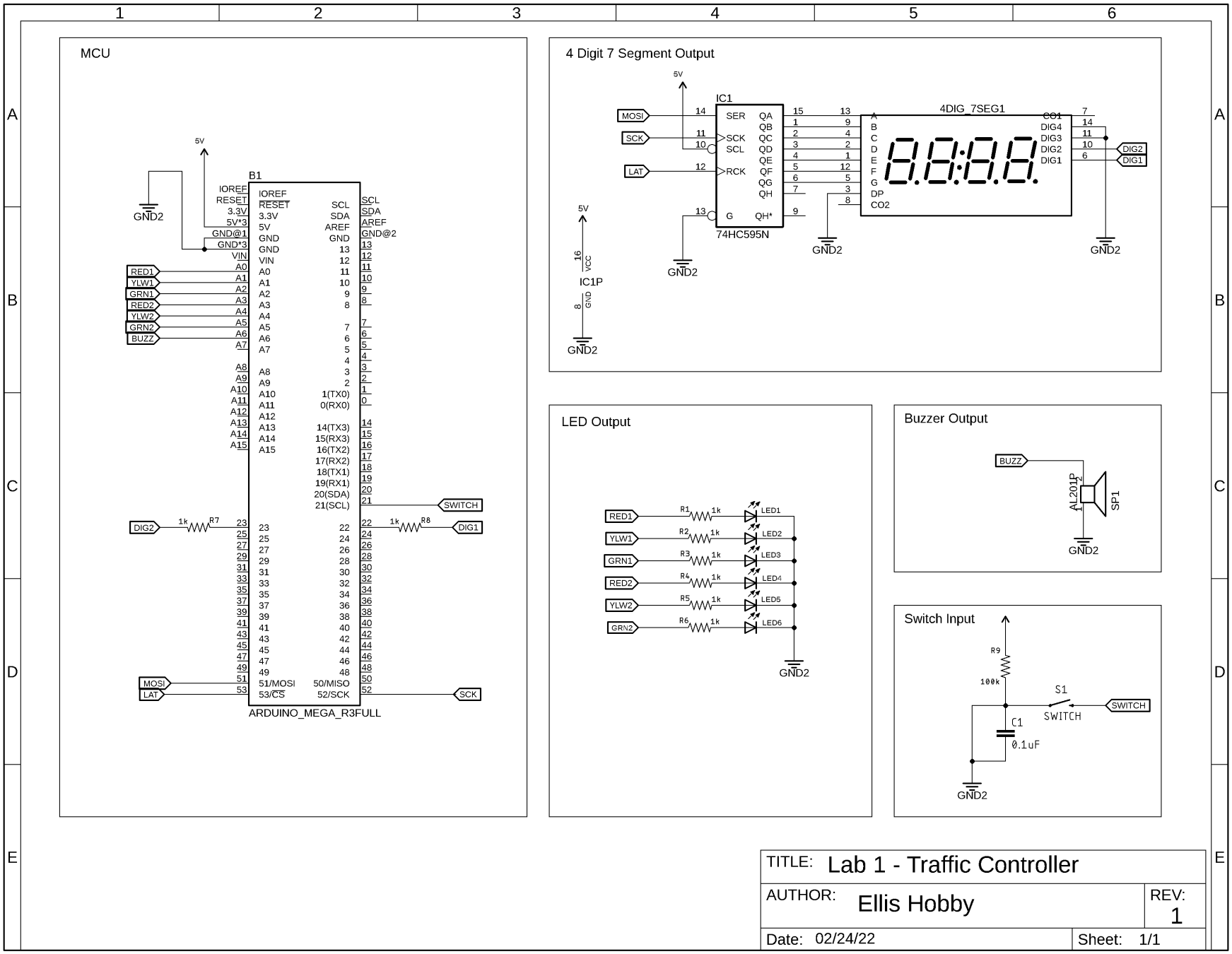
However, it could be said that this MCU is lacking in terms of clock speed. Topping out at 16 MHz may not make it the best choice for high-speed data applications.

Learning how to synchronize the four digit 7-segment display count was a challenge but also a good learning experience. Care must be taken to ensure that the illusion of individual digit displays may be perceived by the end user. Understanding how to properly access data for the digit displays is equally important. For this project data was stored in a *uint16_t* data type, where the lower 8 bits are used for the first digit and the upper 8 bits are for the second most significant digit. Ensuring the data is properly shifted and masked is imperative to ensuring that the proper data is visualized.

Implementation of a theoretical FSM was an import aspect of this project. Drafting the FSM before attempting to implement functional code reinforced the importance of understanding theoretical design constraints and specifications. Furthermore, understanding the forms an FSM based system might take shape was considered, for instance a class-inheritance based method versus a switch-case based method. Considering the size and complexity of the FSM at hand dictates the optimal method of choice. For smaller systems a switch-case based implementations appears to make more sense. Whereas for larger systems a class-inheritance based system appears to be a better choice.

Although the author has a firm grasp on other important aspects of this project, such as timers, digital I/O, and interrupts, putting these features into practice within a physical system is always a good practice exercise. Furthermore, opting to use low level methods for accessing these features, compared to Arduinos function wrappers, may be more time consuming but leads to a greater understanding of the actual bare metal hardware operation.

Appendix


MCU
B1
ARDUINO_MEGA_R3FULL
4 Digit 7 Segment Output
IC1
74HC595N
4DIG_7SEG1
IC1P
LED Output
Buzzer Output
BUZZ
SP1
Switch Input
SWITCH
TITLE: Lab 1 - Traffic Controller
AUTHOR: Ellis Hobby
REV: 1
Date: 02/24/22
Sheet: 1/1

References

[1] Atmel, *8-bit Atmel Microcontroller with 16/32/64KB In-System Programmable Flash,* Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V ed., San Jose, CA: Atmel Corporation, 2014.

[2] T. Instruments, "Debounce a Switch," Texas Instruments Incorporated, Texas, 2020.

[3] Avago, *18:88 and 88:88 0.56" Four Digit GaP HER, ,* Avago Technologies, 2006.

[4] Microchip, "ATmega32u4," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/ATmega32U4.

[5] Microchip, "ATmega328," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/ATmega328.

[6] Microchip, "ATmega2560," Microchip, [Online]. Available: https://www.microchip.com/en-us/product/ATMEGA2560.