0. General Information

Student Name:	Ellis Hobby
Student ID Number:	01928869
Team Name:	Tube Disaster
Team Member Names:	Ellis Hobby
Date of Completion:	04/12/23
Demonstration Method:	Zoom

1. Design

The current project requires the design of an embedded system that can be used to control a fan driven by a direct current (DC) motor. The fan shall run in either direction; clockwise (CW) or counterclockwise (CCW), and have four discrete speed selections; off, 1/2, 3/4, and full. The system shall include a real time clock (RTC) for keeping track of time passed since system startup. A 16x2 liquid crystal display (LCD) shall be used to indicate the current time, fan speed, and fan rotation. Buttons shall be used to change the fan speed and direction. Additionally, a sound sensor shall be used to detect frequency input and change the fan speed according to the following stimulus: consecutive inputs of C4 (262 Hz) to A4 (440 Hz) will cause the fan to speed up. Similarly, a reversed input of A4 to C4 shall cause the fan to decrease in speed. The frequency detection shall be allowed a 2% margin of error, where C4 will then be recognized between the 256.76 to 267.24 Hz frequency range and A4 within the 431.2 to 448.8 Hz range [1]. Full system schematics can be found in the Appendix.

1.1 Hardware Design

The core of the system is built upon the ATmega2560 from Atmel. This 8-bit microcontroller (MCU) has a processing speed of approximately 16 MHz and a multitude of input-output (IO) capabilities that are well suited for the desired system functionality, namely the pulse width modulation (PWM) module, inter-integrated circuit bus (I2C), and analog to digital

converter (ADC). The PWM module allows us to create a square wave with a variable pulse width at some arbitrary frequency within the limits of the system clock (16 MHz). The DC voltage that appears at the output of the PWM is directly proportional to the on-time of the pulse within one cycle [2]. This property can be then used to create a variable voltage source for the fan speed control. The I2C bus is used to manage communication with the RTC module for timekeeping data. The ADC permits analog voltage measurements generated by the sound sensor to be utilized for processing by way of digital conversion. The general-purpose digital input and output (GPIO) pins are used for transmitting display data to the LCD. Additionally, where available, the GPIO can be used to asynchronously interrupt the program for important data input such as fan rotation or speed updates. Figure 1 shows a schematic for the MCU circuit block, the net labels were used to indicate how each pin connects to the external peripherals.

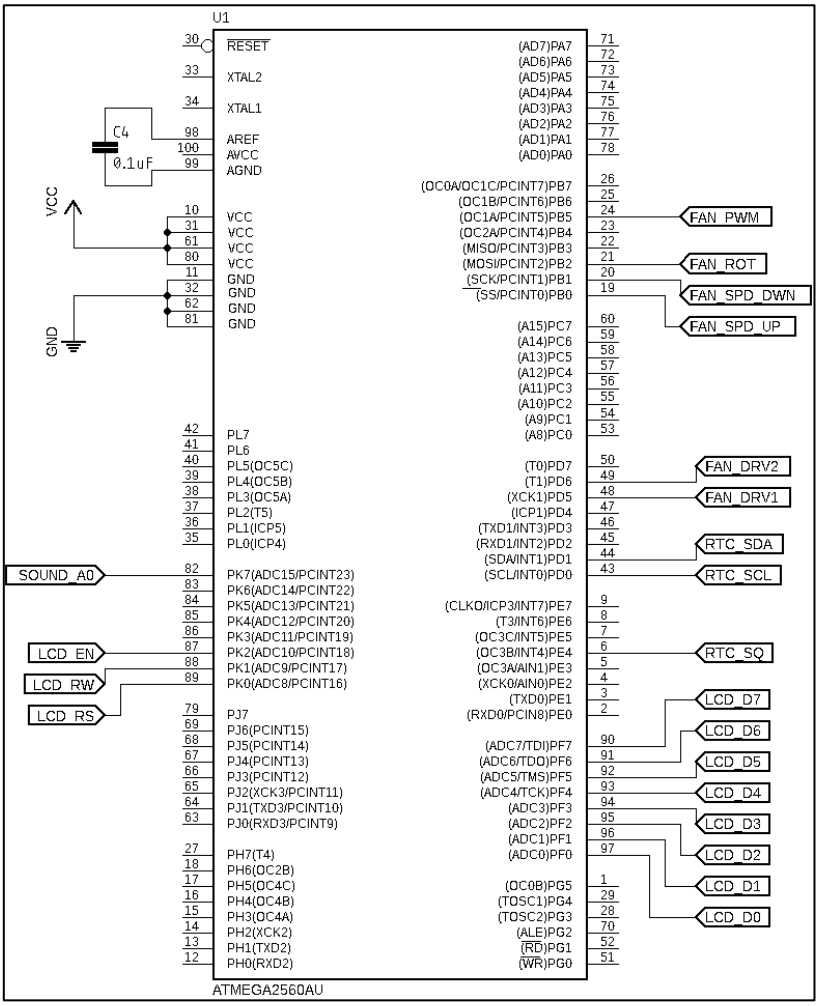


Figure 1: Atmega2560 Circuit Block

1.1.1 Fan Circuit Block

The DC motor, Figure 2, was configured to operate on a dedicated 5V power supply separate from the digital circuitry. This helped to reduce unwanted noise within the digital power lines that may result from electromagnetic interference (EMI) or transient current spikes that occur while running the motor. To interface the digital circuitry for motor control the L293 H-bridge motor driver integrated circuit (IC) was used. This chip features two power supply inputs, one for the internal logic control circuity (V_{CC1}) and one for the dedicated analog motor supply (V_{CC2}) [3]. Figure 3 shows the fan and motor driver circuit block, here MCU was configured to feed two of the driver control inputs 1A and 2A using pins PH5 and PH6. These pins were used to determine how the driver routes the polarity of the motors output power; this enabled the rotational direction to be changed. Pin PB5 was configured for PWM output and connected to enable 1 (EN1) on the L293. This pin turns the output power to the fan on an off. Feeding it with a PWM signal allows us to vary the relative voltage that appears on the outputs, thereby enabling the motor speed to be changed. The motor leads were connected to outputs 1Y and 2Y on the L293. The enable input (1EN) was connected to the PWM signal which pulsed the driver the outputs (1Y,2Y) and activated the motor. The control inputs (1A,2A). Figure 4 depicts a logic diagram provided in the datasheet. This can help visualize how these mechanisms are implemented within the chip and how they may be applied within the current system.

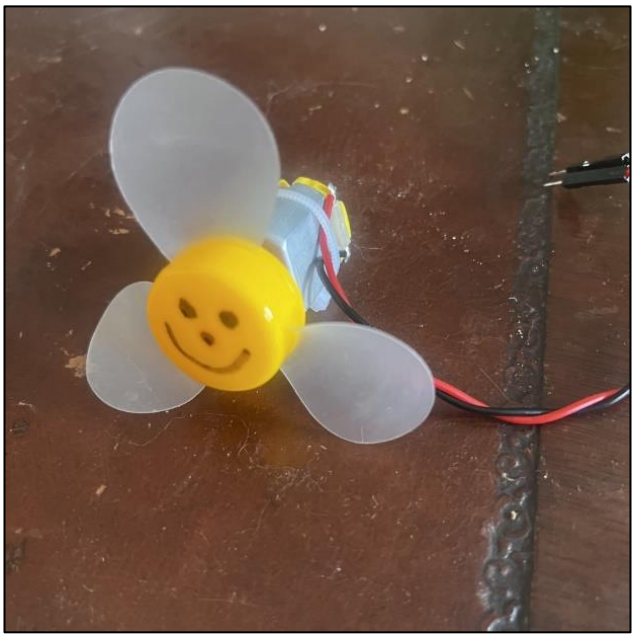


Figure 2: DC Motor with Fan Blade

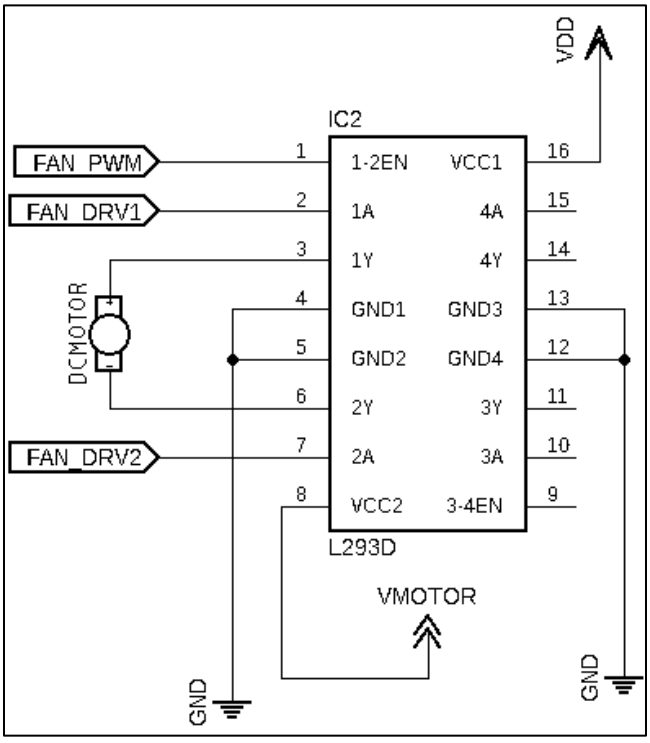


Figure 3: DC Motor with L293 Driver Circuit Block

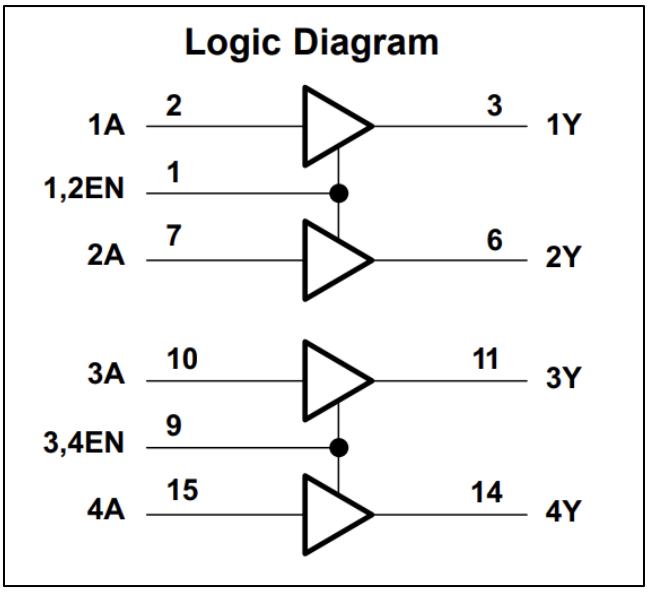


Figure 4: L293 Logic Diagram

Motor control was achieve using three tactile switches for speed up, speed down, and rotation direction, Figure 5. These were connected to MCU pins PB0, PB1, and PB2 respectively, this enabled them to be used as Pin Change Interrupts (PCINT0) for asynchronously updating the motor parameters. To ensure reliable data input each switch line includes a low pass filter for reducing unwanted mechanical noise. Each low pass filter is constructed using a 0.1 μF capacitor and 100 kΩ resistor to create a delay of approximately 10 ms, as described by Equation (1) [4].

$$\tau = R * C \quad (1)$$

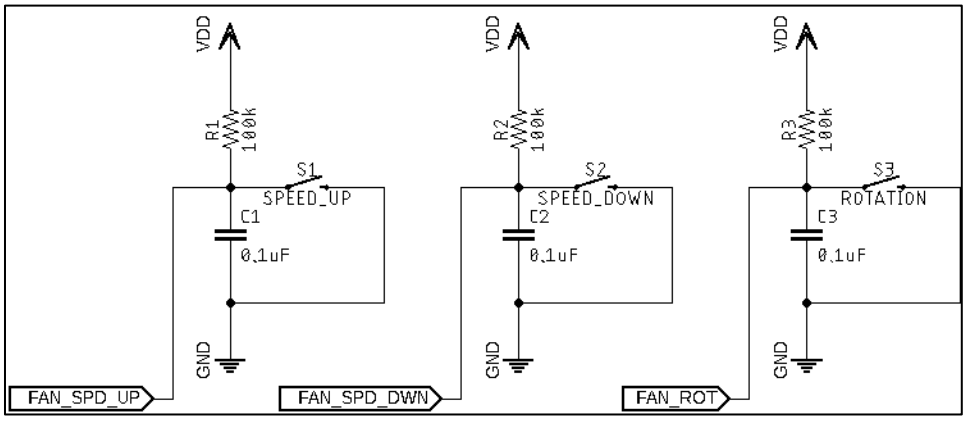


Figure 5: Fan Switch Control Circuit Block

1.1.2 Sound Sensor Circuit Block

The Sound Sensor module, Figure 6, was used as an additional control mechanism for motor speed control. As described above a sequence of C4-A4 was used as speed up command and A4-C4 for speeding down. Figure 7 shows the Sound Sensor circuit block, here the analog output of the Sound Sensor (A0) was connected to ADC15 (PK7) on the MCU. This enabled the signal to be converted into the digital domain for frequency analysis. The module provides a trim pot for adjusting the gain of the built-in amplifier. It is important to properly set this so that the sensitivity will not be too high or low. Additionally, understanding the output voltage range determines the proper reference voltage for the ADC. The chosen setting for this was achieve using an oscilloscope to determine peak voltage values, Figure 8.

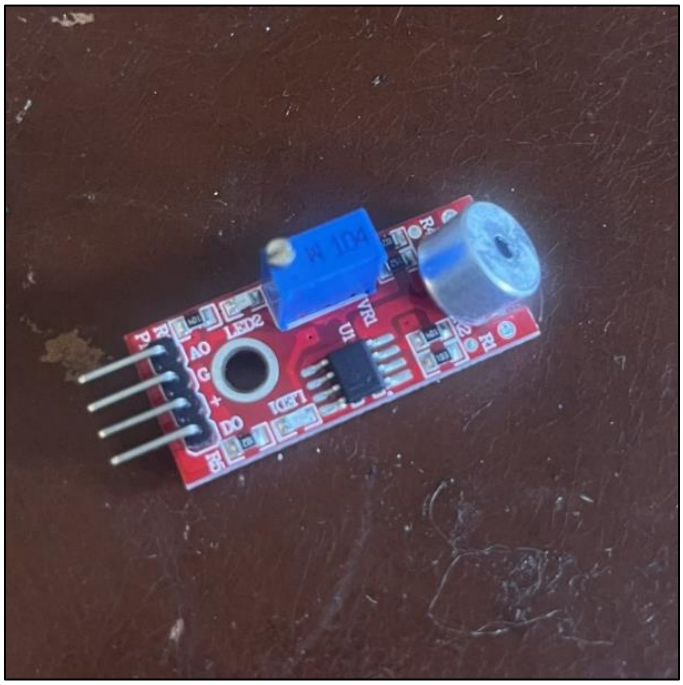


Figure 6: Sound Sensor Board

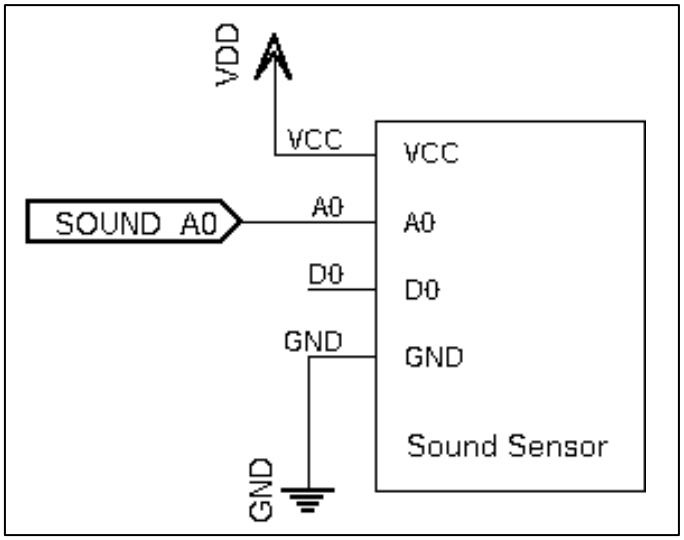


Figure 7: Sound Sensor Circuit Block

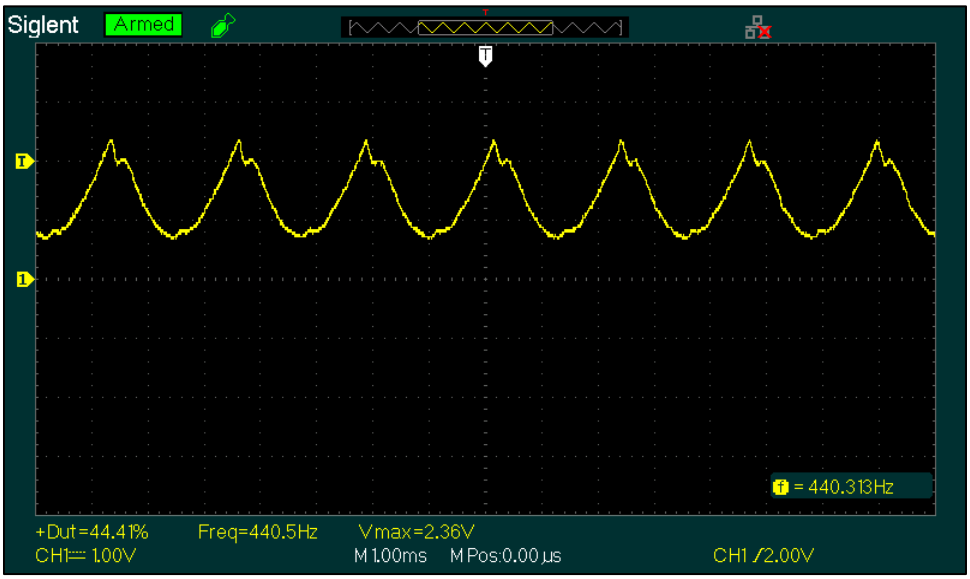


Figure 8: Sound Sensor Input Oscilloscope – A4 (440Hz) Input

1.1.3 RTC Circuit Block

The DS1307 RTC module was used for time keeping, Figure 9. This device includes a battery back up to ensure that the time passage is still tracked when the main power supply is removed. The internal clock/calendar tracks seconds, minutes, hours, day, data, month, and year [5]. This information is accessible by accessing the devices internal storage registers using the I2C communication protocol. The clock is configurable to 24-Hour or 12-Hour with AM/PM format by writing to its internal control registers. The current system uses a 24-Hour formatting scheme. Additionally, the device offers a square wave output pin that is based on the internal clock. This square wave may be set to 1 Hz, 4.096 kHz, 8.192 kHz, or 32.768 kHz which enables external devices to directly synchronize with it. Figure 10 shows the circuit block for the RTC module within this system. The serial clock (SCL) and serial data (SDA) lines were routed to MCU pins PD0 and PD1 respectively. The square wave output was configured to 1 Hz and connected to PE4 which was used for external interrupts (INT4) from the RTC.

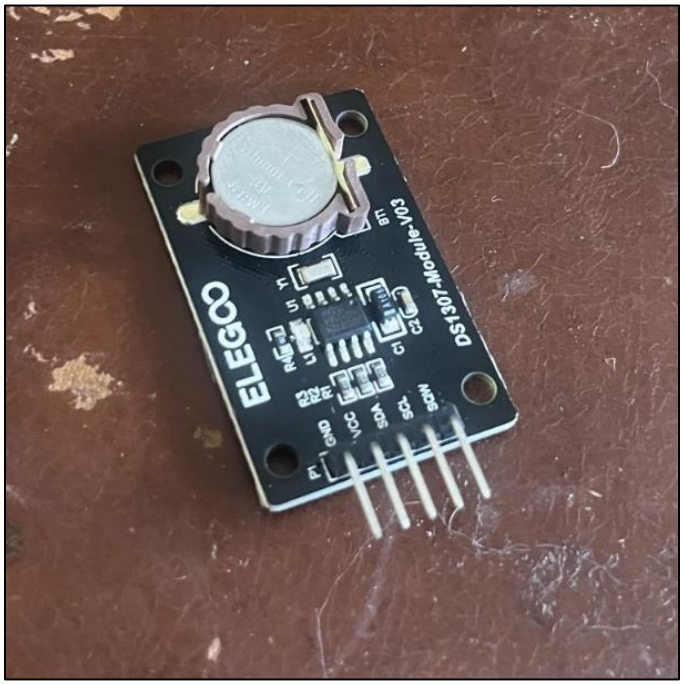


Figure 9: DS1307 RTC Module

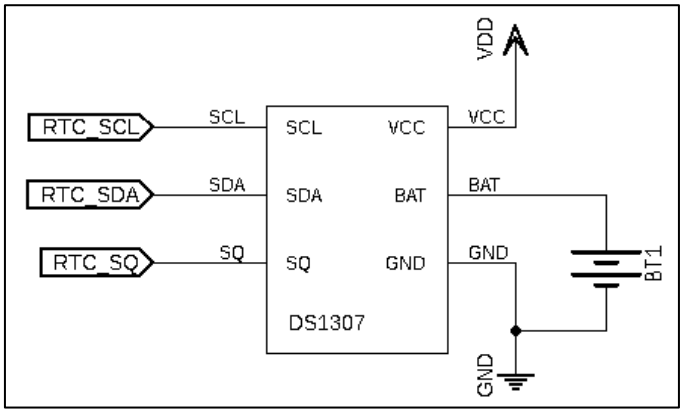


Figure 10: RTC Circuit Block

1.1.3 LCD Circuit Block

An LCD screen was used to display the current time, the fan speed, and the fan rotation direction. The module, Figure 11, is in a 16x2 format, where 16 is the number of characters per line and 2 is the number of lines. For interoperating screen control data, the module uses the Hitachi HD44780U LCD driver. This driver accepts parallel data input on pins D0-D7 as bytes (8-bit) or nibbles (4-bit) [6]. The current application uses byte communication, therefore, all of PORTF (PF0-PF7) on the MCU was reserved for sending LCD data messages. Three additional control inputs are required for interpreting values on the data bus; register select (RS), read/write select (RW), and enable (EN). These were connected to PK0, PK1, and PK2 respectively. For control over the screens contrast a 10 kΩ potentiometer was connected with its wiper to pin V0 on the LCD and its inputs connected to +5V and Ground. Figure 12 shows the layout for the LCD circuit block within the system schematic.

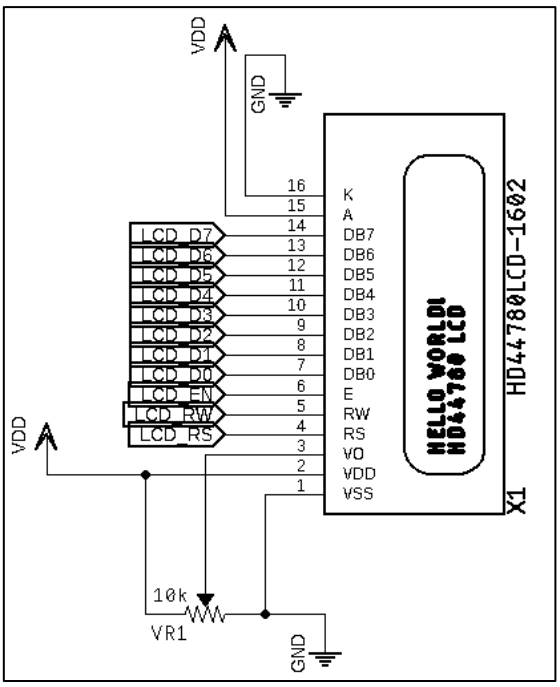


Figure 12: LCD Circuit Block

Figure 11: LCD Module

1.1.4 Power

The system was built using three separate power sources with a common ground: one for the MCU, one for the digital peripherals, and one for the motor. This became necessary due to transient noise artifacts introduced onto the power lines while running the motor. Digital circuitry can be very sensitive to noisy power lines, so separation of analog and digital supplies was necessary. The MCU receives power from the universal serial bus (USB) port. The digital peripherals and motor received power from an Elenco XP-720 bench top power supply, Figure 13. This supply offers three output sources: +5V at 3A, + 0 – 15V at 1A, and - 0 – 15V at 1A. The +5V source was connected to the digital peripherals. The variable +15V source was set to +5V and used for the motor. It should be noted that the MCU was given separate power solely due to extra sources being available, typically it would be fine to connect it with other digital circuity.

Figure 13: Elenco XP-720 Power Supply

1.2 Software Design

The software design implements four custom data structures dedicated to performing the necessary functionality for each of the main peripherals (Fan, Sound Sensor, RTC, LCD), and one additional data structure for managing I2C transactions with the RTC module. All programming was done using low level AVR register manipulation in an effort to optimize processing time. Each modules class implementation was tested on its own with hardware to verify the expected behavior. This also enabled class specific bugs to be caught and fixed early on before complexity was increased during the integration stage. The test programs are included along with the main program in the projects GitHub repository. This is currently private with collaborator access only, to request access please email the author at: ellis_hobby@student.uml.edu.

1.2.1 Fan Control Data Structure

The class in charge of managing fan/motor operation was named *PWM1*. The header file for this class defines the necessary pins and port access for all GPIO related to the fan functionality. During class initialization the two motor driver pins and PWM pin are configured as outputs. To generate the PWM signal *Timer1* was configured to *Fast PWM Mode* with 9-bit resolution. Its frequency is dependent on the clock prescaler value (*N*) and resolution (*TOP*) as determined by Equation (2) [7]. Using a value on 64 for the prescaler (*N*) and given 9-bit resolution (*TOP=511*) results in a PWM frequency of approximately 480 Hz. The three fan control switches are set digital inputs with internal pull-up resistors enabled and PCINT0 interrupts enabled.

$$f_{pwm} = \frac{f_{cpu}}{N(1 + TOP)} \quad (2)$$

The PWM signal essentially works in the background once configured correctly. Its theory of operation relates to the value loaded into the output compare registers (OCR1A). Every cycle of the timer clock (f_{cpu} / N) increments the timer count register (TCNT1). When the count of

TCNT1 reaches the value loaded into OCR1A the PWM output pin is set HIGH. The counter keeps incrementing until it reaches the TOP value and rolls over to 0x00 (BOTTOM). Upon reaching BOTTOM the PWM output is set LOW. To vary the pulse-width (on-time) we must load different values into OCR1A. Figure 14 shows a timing diagram included in the ATmega2560 datasheet which helps to visualize this process. Equation 3 shows OCR1As value can be determined to output a desired pulse-width percentage. For example, if the desired pulse-width was 25% then OCR1A must be loaded with a value of 128 ((TOP+1) * 0.25 = 512 * 0.25) . The speed requirements of the fan for this system are off (0%), 1/2 (50%), 3/4 (75%), and max (100%). Knowing this we can then determine that OCR1A must be loaded with either 0, 256, 384, or 512 depending on the requested fan speed.

$$\boldsymbol{OCR1A = (TOP + 1) * PW} \quad \textbf{(3)}$$

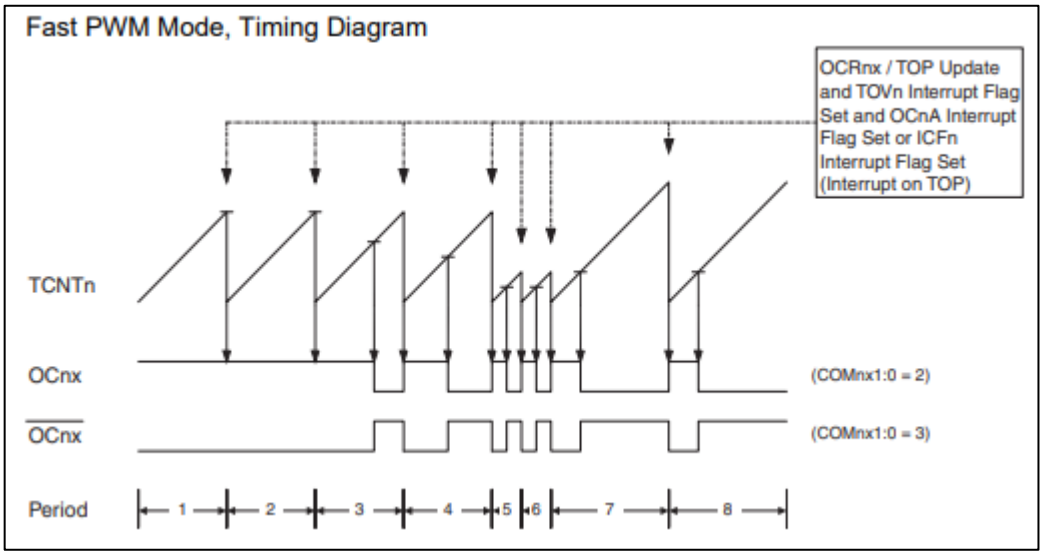


Figure 14: Fast PWM Mode Diagram [7]

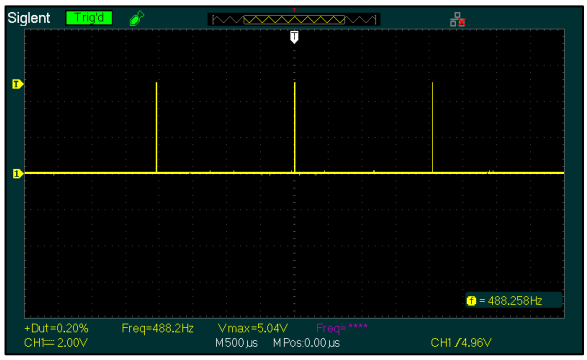


Figure 17: Off PWM - 0%

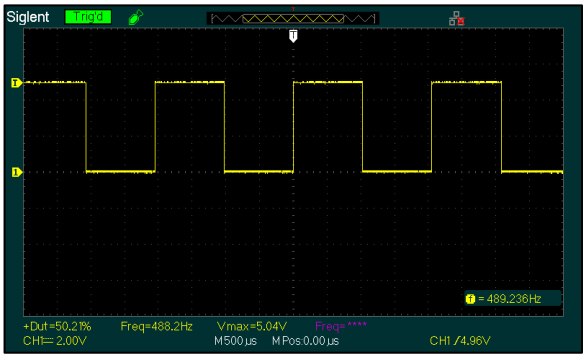


Figure 18: 1/2 PWM - 50%

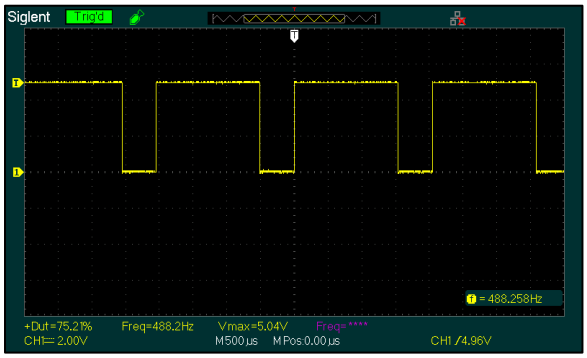


Figure 16: 3/4 PWM - 75%

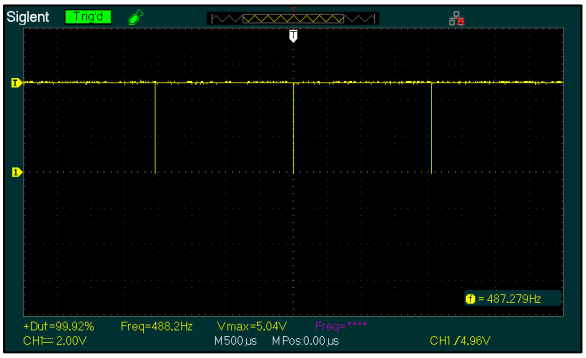


Figure 15: Max PWM - 100%

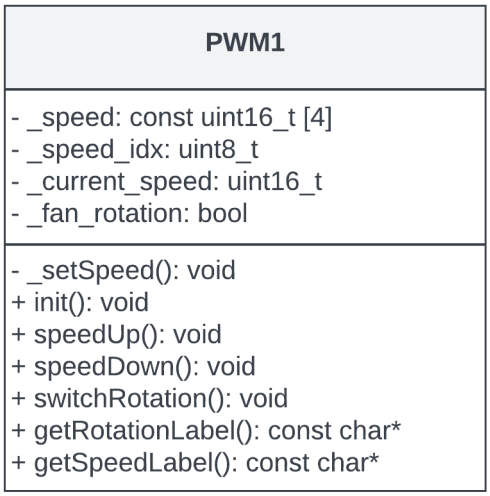


Figure 19: PWM1 UML Class Diagram

The four values required for each fan speed were stored into a private class member array called *_speed*. Putting them in an array like this made constructing functions for speed control much easier as only two would be needed, *speedup( )* and *speedDown( )*. Within these functions a private data member for tracking (*_speed_idx*) can simply be incremented or decremented, modded with a value of 4 to stay within array bounds, then used to index the *_speed* array and set the value of OCR1A. To switch the fan rotation a class method called *switchRotation( )* was created which simply inverts the values of the two motor driver pins. Two additional methods, *_getSpeed( )* and *_getRotation( )*, were created to return string values regarding the current state of speed and direction for the LCD display.

The three switches used for setting the fan speed and direction all jump to the same interrupt vector. This is the nature of how Pin Change Interrupts work. This required checks nested within the interrupt service routine (ISR) to determine which switch was pressed and achieved by saving the last state of the relevant port (PORTB). Using the known state allows us to perform an exclusive-or (XOR) with the current state which results in a value indicating which port pin has changed. This value can then be checked against bit masks for each switch. Once a match has occurred the relevant switch function can be dispatched.

1.2.2 Sound Sensor Data Structure

The class in charge of managing acquisition and processing of data from the Sound Sensor was simply named *SoundSensor*. This class determines the fundamental frequency of a sample set using Fast Fourier Transform algorithms (FFT) provided in the *arduinoFFT* library. A total of 128 samples are taken per set at a sample rate of 880 Hz. This conforms to the Nyquist criterion which states that the sampling rate must be double the highest frequency of interest (A4, 440 Hz) to guarantee an accurate discretized representation [8]. The samples are taken using ADC15 (PK7) and the sampling rate is determined by *Timer4*.

Upon initialization of this class an *arduinoFFT* object is created and saved as a private class member named *_FFT*. Next the ADC multiplexer (ADMUX/ADCSRB) is set to use ADC15 as an input, the reference voltage is set for 2.56V, and the conversion rate is set for 125 kHz. Then the timer is setup to operate in Clear Timer on Compare Match (CTC) Mode and count at a rate of

16MHz (f_{CPU}) determined by a prescale value of 1. In CTC mode the timer will count up until it reaches the value stored in OCR4A at which point the output compare interrupt flag (OCF4A) will be set and the count will reset to zero. Changing the value stored in OCR4A causes a change in frequency of the resulting interrupts. This behavior was used to set the sampling rate to the desired 880Hz.

To analyze the frequency content of sound input the class method *senseFreq( )* is called. This method takes 128 samples from the ADC and stores the resulting values into a private class array name *_real[128]*. In order to maintain a sampling rate of 880 Hz the function waits for the OCF4A flag to be set between each reading. Once the full sample set has been obtained the *_FFT* object is used to perform frequency analysis which determines the fundamental frequency of the sample set. The resulting value is then saved to a private class member named *_this_freq* and compared to the previous frequency reading which is stored in *_last_freq*. If a pattern match of A4-C4 or C4-A4 occurs between the two a class variable named *_seq* is set to either *SEQ_UP* (1) or *SEQ_DWN* (2), otherwise it is set to *SEQ_NONE* (0). The main program loop first calls *_senseFreq( )* and then obtains the resulting sequency by calling *getSeq( )*. The returned value is compared to *SEQ_UP* and *SEQ_DWN*, if a match occurs the relevant fan control method is called.

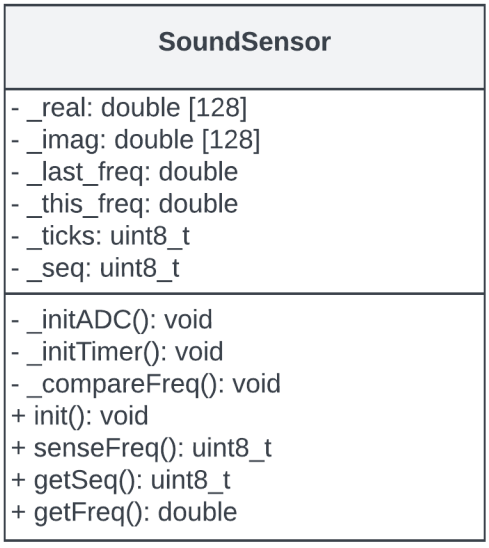


Figure 20: RTC UML Class Diagram

1.2.3 RTC Data Structure

To synchronize I2C transactions with the DS1307 a utility class named *I2C* was created. This class by default is initialized with a bit rate of 400 kHz. It includes the appropriate methods for start, stop, read, write, and slave access conditions through low level register access as defined in [9]. Figure 21 shows an example of an I2C transaction taking place with the DS1307 viewed on an oscilloscope. Here the yellow trace is the data line (SDA) and the purple trace is the clock line (SCL).

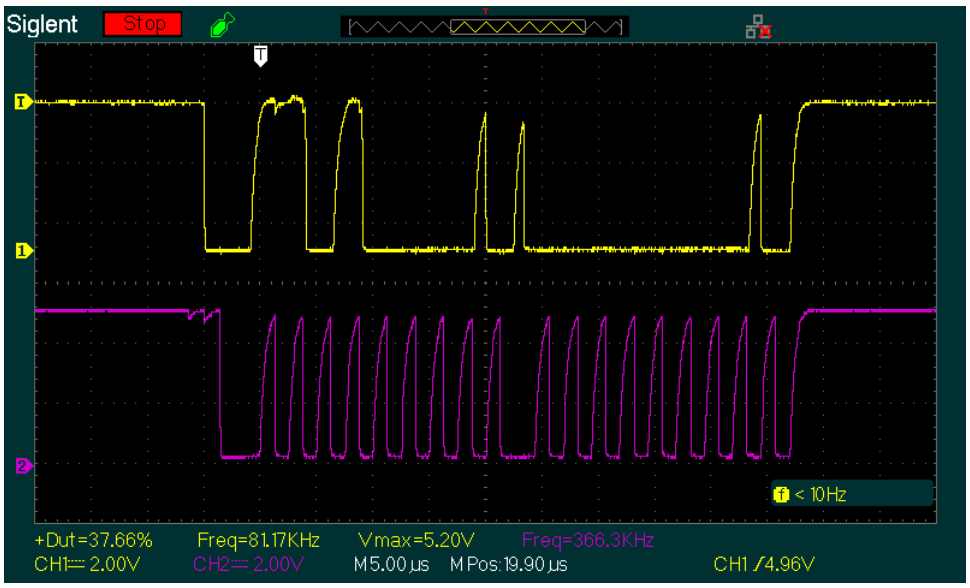


Figure 21: I2C Transaction with DS1307

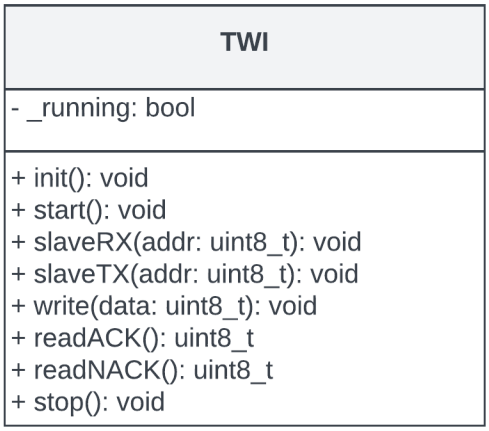


Figure 22: TWI UML Class Diagram

To coordinate data transfer with the DS1307 module a class called *RTC* was created. The class header file (*rtc*.h) defines the DS1307 register addresses and I2C address as specified in the datasheet [5]. It also defines two utility structures that are used by the *RTC* class. The first is called *ControlRegister* which is a union-struct bit-field used for tracking the state of the internal DS1307 control register. The second is a struct called *TimeKeeping* which collects the three values of time that are being tracked, seconds, hours, and minutes.

When creating an *RTC* class object it must be passed and *I2C* object which it stores as a class member to use for data transactions. During initialization the square wave output of the DS1307 is configured to pulse at a frequency of 1 Hz by writing the appropriate value to its internal control register. Then PE4 on the MCU is set to a digital input and enabled for external interrupts (INT4). The square wave output from the DS1307 is routed to PE4, this enables the MCU to be signaled once per second that the time has updated and should be read accordingly.

The main program loop requests the current time by calling *getTime( )* on an *RTC* object. This method performs the necessary steps to address, access, and read data from the DS1307. The data received will not directly read as the correct decimal value due to how it is stored within the DS1307 registers and therefore must be converted before it is saved. Table I shows the internal register map for the device [5]. Here we can see how the time values are stored by magnitude rather than their actual decimal value. For example, in the seconds register the tens place is stored in bits 4-6 and the ones place is stored in bits 0-3. Consider if the binary value 01010111 was stored in this register, which represents 87 in base 10. Clearly it would not make sense to try to use a value of 87 for seconds as they only range from 0-59 on a clock. Applying the appropriate conversion described by Equation 4 results in a value of 57 seconds. Equations 5 and 6 are the conversions used for minutes and hours respectively. Once the all the raw data for seconds, minutes, and hours has been read and appropriately converted, it is returned to the caller as a *TimerKeeping* data structure for ease of access.

$$sec_{conv} = \Big(\big((data \gg 4)\ \&\ 0x70\ \big) * 10\Big) + (data\ \&\ 0x0F) \quad \textbf{(4)}$$

$$min_{conv} = \Big(\big((data \gg 4)\ \&\ 0x70\ \big) * 10\Big) + (data\ \&\ 0x0F) \quad \textbf{(5)}$$

$$24hr_{conv} = \Big(\big((data \gg 4)\ \&0x30\ \big) * 10\Big) + (data\ \&\ 0x0F) \quad \textbf{(6)}$$

Table I: DS1307 Internal Register Map *[5]*

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00h	CH	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/ AM							
03h	0	0	0	0	0	DAY			Day	01–07
04h	0	0	10 Date		Date				Date	01–31
05h	0	0	0	10 Month	Month				Month	01–12
06h	10 Year				Year				Year	00–99
07h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08h–3Fh									RAM 56 x 8	00h–FFh

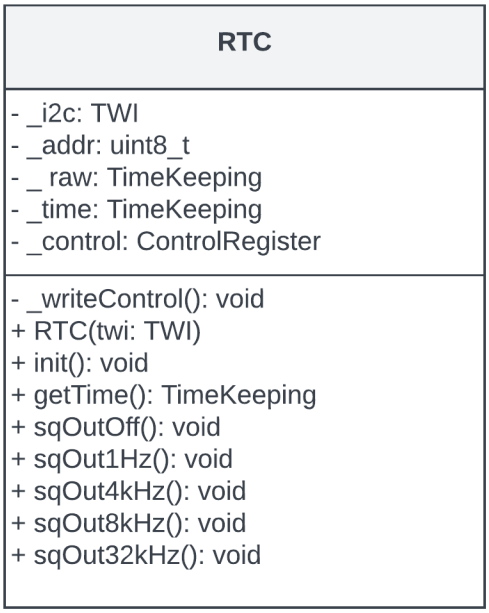


Figure 23: RTC UML Class Diagram

1.2.4 LCD Data Structure

Facilitating display data was achieved using the custom *LCD* class which includes methods for controlling and writing to the Hitachi HD4470 driver. Additionally, six union-struct bit-fields were created to map the driver's internal instruction registers: *_entry_mode*, *_display_control*, *_functionality*, *_cgram_addr*, and *_ddram_addr*. The field bits were mapped using Table II, given

initial values per the data sheet [6], and written to the device at run time. Any following changes requested by the main program could then be tracked and changed without having to read from the driver. This allowed for complex functionality in terms of control. Although much of the implemented control mechanisms were not used in this project they were still coded purely out of interest and the potential of reusability. The most important control used in this project was the ability to write addresses to data display RAM (DDRAM) enabling the cursors' location to be moved. This allowed the information on the display to be formatted in an aesthetically pleasing and readable manner as depicted in Figure 25.

Table II: DS1307 Internal Instruction Registers *[6]*

	Code										
Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.

Figure 25: LCD Formatting

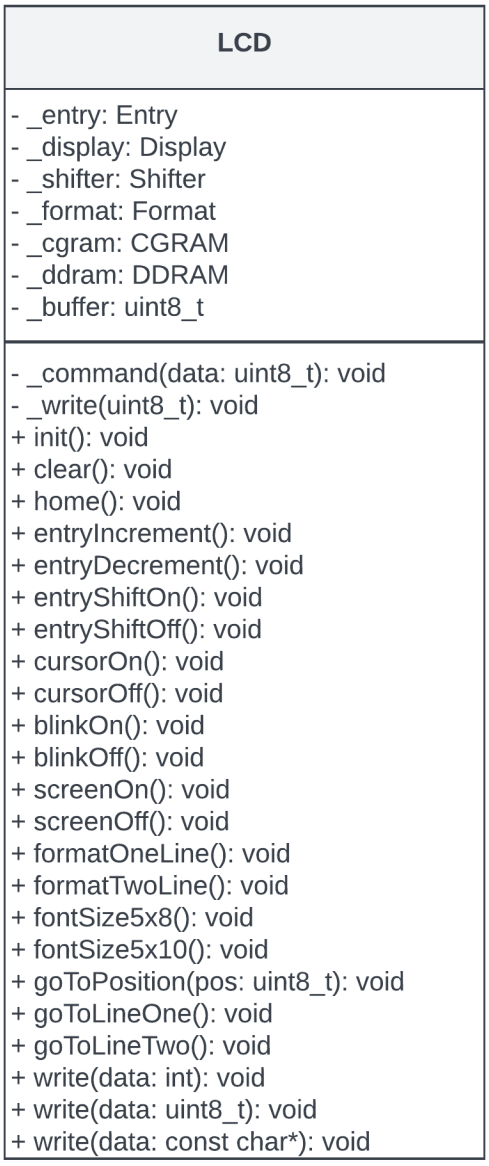


Figure 24: LCD UML Class Diagram

Upon initializing the *LCD* class the data (D0-D7) and control (RS,RW,EN) pins are set as digital outputs and defaulted Low. A small delay of 100ms is given to allow the internal electronics on the driver time to warm up and then commands are written to turn the screen on, enable two-line display, turn the cursor/blink off, and set the font size to 5x8 pixels. The two class methods used by the main program loop are *write(const char* data)* and *gotoPosition(uint8_t pos)*. The *write* function accepts a string and writes it to the screen. When it encounters a string terminator (NULL, 0x00) it returns without writing to prevent unintentional blank spaces. The *gotoPosition* function accepts a position value and sets the cursor to that position on the screen by writing the appropriate address to the DDRAM. Valid *pos* arguments are values 0-30, where 0-15 is each position on the first line, and 16-30 is each position on the second line. It is important to note here that each line spans 40 characters although they are not displayed, therefore an offset of 40 must be applied to positions over 15 to obtain the correct DDRAM address. The *gotoPosition* function is used internally during write functions as a way to recognize end of line character (EOL) such as " */n* ". When an EOL character is encountered the *LCD* class sets the cursor to the next instead of writing to the screen. If the current position is within the first line range, it will set the position to the second line and vice versa. This is done by checking the DDRAM address value stored within its bit-field.

1.2.5 Main Program Loop

The main program loop begins by including the necessary files to create the objects described above. It also defines ISRs for the fan switches and RTC square input vectors. The fan switch control ISR works as described above. The RTC square wave ISR calls the *getTime( )* on the *RTC* object to request the current time. It was thought that containing these here would clarify their usage within the context of the main program rather than having them defined within their relating source files. Once all objects are initialized it enters the main loop.

At the start of each loop the Sound Sensor frequency is read and compared to the up/down sequence values (*SEQ_UP / SEQ_DWN*). If a sequence match occurs the corresponding motor control function is called. At the end of the loop a function is called to update the screen with the most recent system information (*updateLCD( )*). Within this function interrupts are disabled to

prevent multiple data access. The clock values are then buffered, converted into a string formatted as “00:00:00”, and written to the LCD starting from position 0. Next the fan rotation is requested and buffered into a string with the format of “R:CCW”. The cursor position is changed to 10 and the fan rotation string is written to the screen at the end of line one. Lastly the fan speed is requested, buffered into a string, and formatted as “S:MAX”. The cursor position is changed to 25 and the speed label is written to the end of line two. Interrupts are then reenabled before returning to the main loop. The described loop executes indefinitely until the processor is powered off.

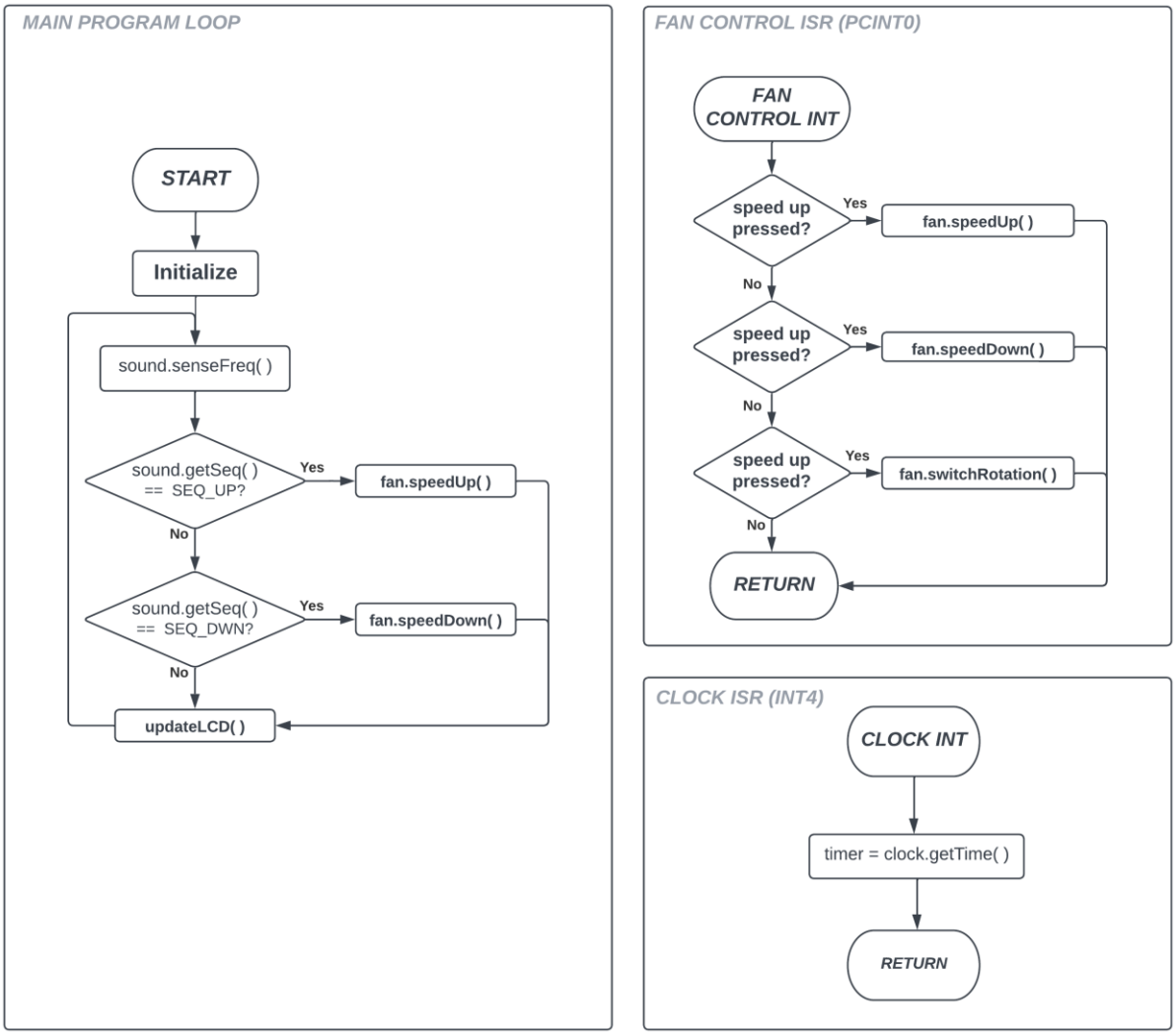


Figure 26: Main Loop Flow Chart

1.3 Results

The complete system prototype was constructed on a breadboard as shown in Figure 27. Colored jumper wires were used as a means to clarify signal nets for troubleshooting purposes. To further assist in the build organization each circuit block was partitioned to a separate area, Figure 28. The L293 and motor were placed on the upper right corner, it was thought that keeping them as far away from the other circuitry as possible would help reduce noise being introduced onto the digital power lines. All circuit blocks were connected and tested with the MCU individually and bugs were identified and fixed. After each circuit was verified to perform as expected the system was integrated as a whole, tested, troubleshot, and fixed. The system generally functioned as expected with some minor bugs that could easily be remedied given more time.

Improvements could be made if this system were to be actually implemented under the same specifications. The most important one would be further isolation of digital and analog power. Although many measures were taken to prevent noise artifacts there was still some presence on the digital supplies as observed on an oscilloscope. A full-fledged design would implement this by using proper ground plane techniques on a printed circuit board (PCB) and connecting analog and digital ground at only one point. Unfortunately, due to the nature of bread boards much stray capacitance is introduced and this sort of noise is somewhat unavoidable despite taking measures such as the ones discussed. Although the noise presence did not appear to cause any immediate issues it would be bad design practice to leave it as such in an actual product implementation.

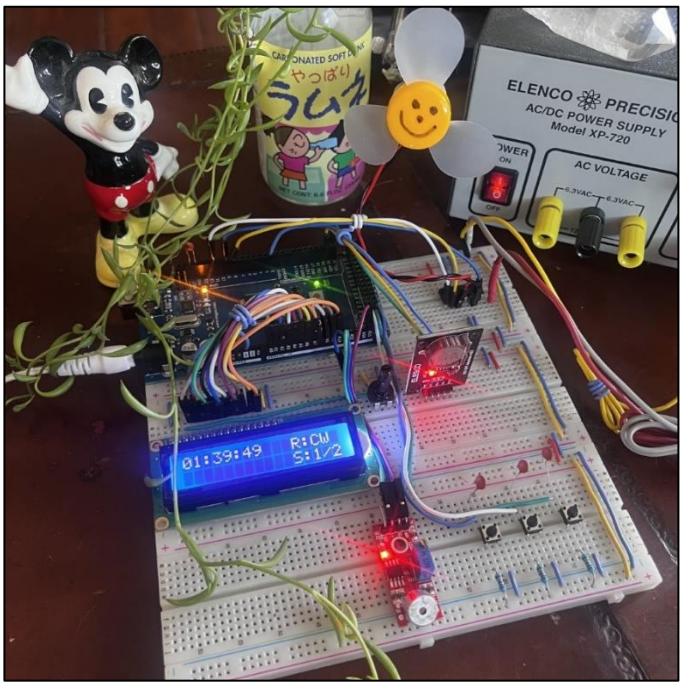


Figure 27: Fan Controller Breadboard Prototype

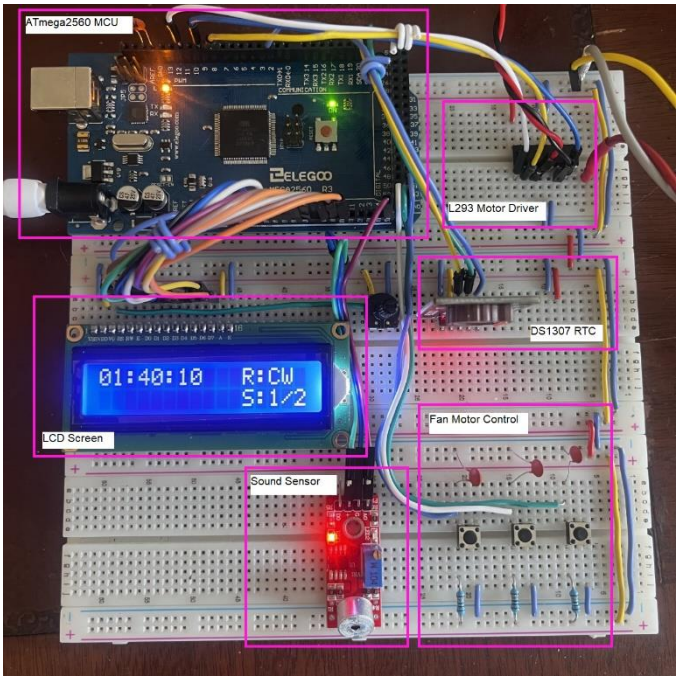


Figure 28: Breadboard Circuit Blocks

2. Problems Encountered

The biggest problem encountered related to implantation of the sound sensor, frequency analysis, and sequence detection. Dialing the sound sensor into the correct sensitivity became a delicate balance of trying to have the input gain high enough to detect sound that were needed while also trying to eliminate extra room noise from accidently being recognized as points of interest. Unfortunately, the current system does seem to suffer from this to some degree. Random noise room noise sometimes causes the system to think a frequency pattern has occurred resulting in the fans motor speed being changed at times when it was not intended. This bug could potentially be solved in two ways. First increasing the number of samples taken for a set would likely promote a more accurate frequency analysis. Currently 128 samples are taken, this could possibly be increased to something like 1024 or 2048. Second the mechanism for saving and comparing frequencies relies on waiting an amount of time before setting *_this_freq* to zero. This could possibly be tuned to ensure the amount of time is within reason and is currently controlled by incrementing a counter (_ticks) each time *senseFreq* is called. If *_ticks* reaches a value of 1000 (currently arbitrarily chose) it will reset itself and *_this_freq*. The purpose of this waiting time is to allow tones in a pattern to experience a small delay whilst still being recognized. If the *SoundSensor* class were to simply set *_this_freq* to zero every time a frequency match had not occurred, it would be hard to identify a relevant pattern if the two tones have even a fraction of a second of delay between them.

Another problem encountered related to developing the *LCD* class. Although a reliable solution was eventually found, getting the EOL behavior to function became a real challenge. This had to do with how the DDRAM address was being tracked and how *setPosition( )* handled changing it. After some critical thinking and reviewing the data sheet the behavior was verified to work as expected. Unfortunately, the EOL feature is not used in the current project. However, it can potentially be reused in other projects.

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 Lab2 directory 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 Learned

Implementing PWM functionality using the ATmega2560s timer modes takes some careful planning and configuration. Once properly setup the device enables the user to generate PWM signals with a high threshold of accuracy. Having a firm understanding on how the Output Compare Registers (OCRnx) affect the generated pulse-width is integral to producing the desired waveform. Furthermore, it is important to realize the resolution being used, as dictated by the Waveform Generation Mode bits (WGMnm). The *Fast PWM Mode* can be configured to operate in one of three different resolutions: 8-bit, 9-bit, or 10-bit. Therefore, care must be taken to understand how the TOP value is affected by each resulting in order to calculate the desired pulse-width ratios. The current project uses the *Fast PWM Mode*, however it should be mentioned that the MCU also offers two other modes: *Phase Correct Mode* and *Phase / Frequency Correct Mode*. These two modes of operation offer higher accuracy waveforms at lower frequencies at a cost of higher complexity of configuration. The difference compared to the *Fast PWM Mode* is that they operate using a dual-slope counter. This means that they count up to the value set by OCRxn and then count down to BOTTOM and repeat. This makes calculating the correct values to load into OCRxn slightly more complicated. Given that the PWM in the current project only operates at a frequency of 488 Hz, one of the other two modes may have been more suitable. However, using *Fast PWM Mode* seemed to work well enough for the specifications.

Most of the other concepts relating to the MCU have been previously investigated but were further reinforced by the current projects increased complexity. For the LCD and RTC modules it was interesting to see how both serial and parallel data communications methods can be used to access control registers on external peripherals. For the Sound Sensor exploring how taking sample sets from the ADC can be used to perform frequency analysis. Most of the custom classes written for this project were done in such a way that they can be used in future projects and make use of much of the functionality offered by the devices. The author is curious to further investigate and improve the LCD class. Specifically, it would be interesting to attempt access and writes to CGRAM control registers which can allow custom characters to be written and displayed.

References

[1] Y. Luo, Lab 3: Controlling a Fan, Lowell: University of Massachusettes Lowell, 2023.

[2] E. Williams, "Chapter 10. Pulse-Width Modulation," in *Make: AVR Programming*, Sebastopol, Maker Media, Inc., 2014.

[3] T. Instruments, L293x Quadruple Half-H Drivers, Dallas: Texas Instruments Incorporated, 2016.

[4] T. Instruments, Debounce a Switch, Dallas: Texas Instruments Incorporated, 2020.

[5] M. Integrated, DS1307 64 x 8, Serial, I2, Sunnyvale: Maxim Integrated Products , 2015.

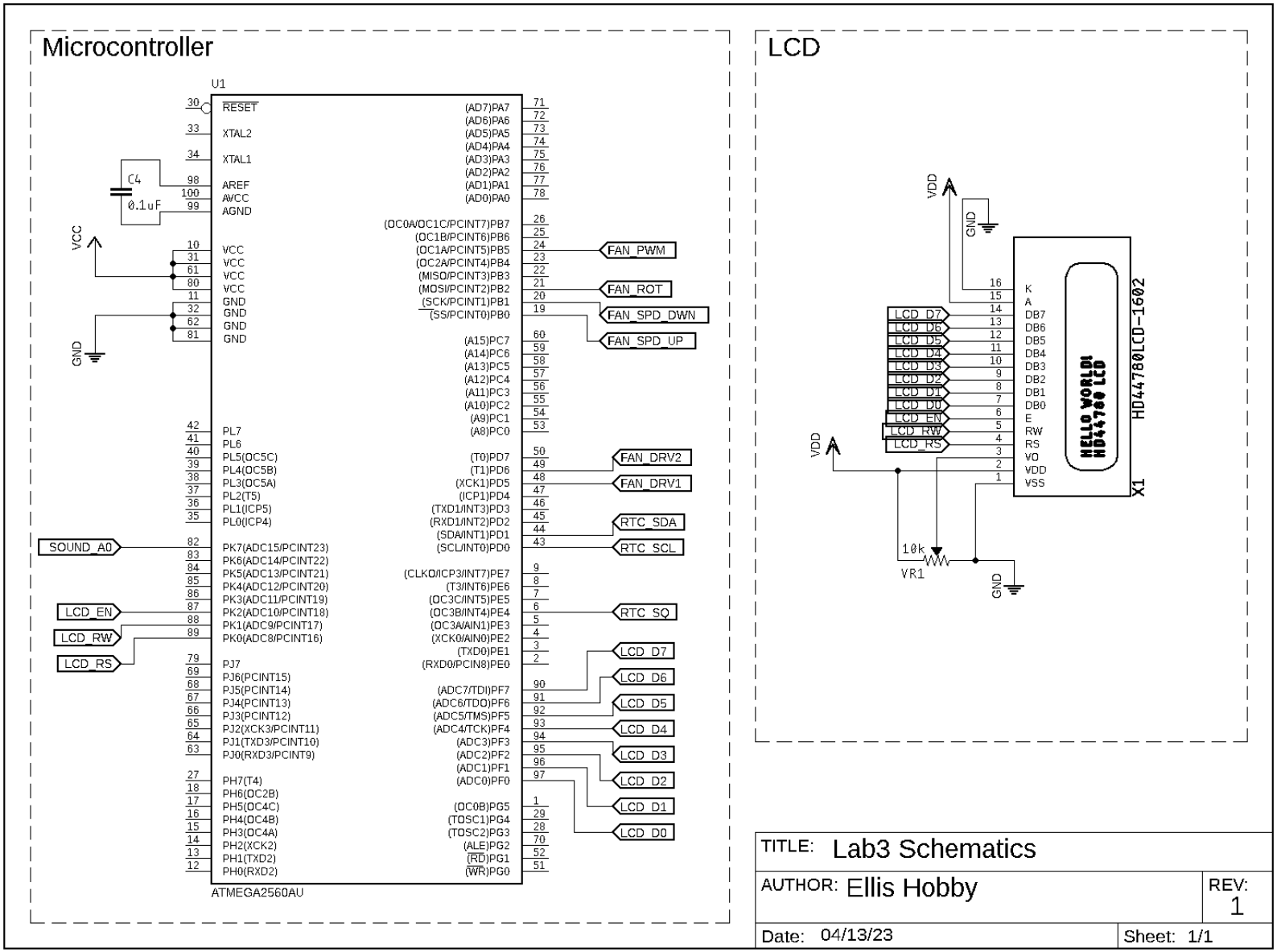
[6] Hitachi, HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver), Japan: Hitachi, Ltd, 1998.

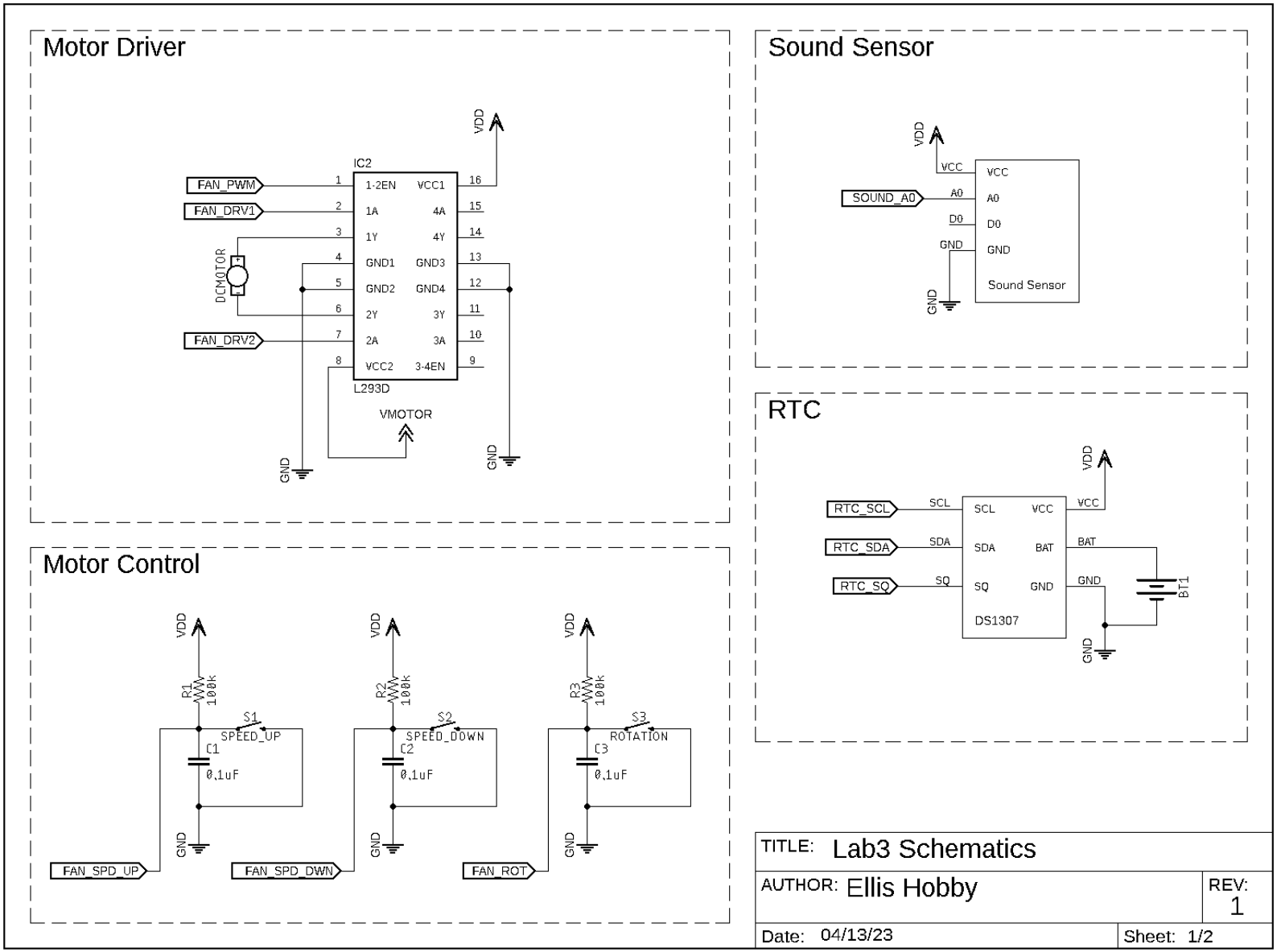
[7] Microchip, "ATmega640/V-1280/V-1281/V-2560/V-2561/V - Complete Datasheet," in *16-bit Timer/Counter (Timer/Counter 1, 3, 4, and 5)*, Microchip Technology, 2000, pp. 133-163.

[8] M. Russ, "1.12.7 Sample rate," in *Sound Synthesis and Sampling, 3rd Edition*, Burlington, Focal Press, 2008.

[9] Atmel, "2-wire Serial Interface," in *ATmega640/1280/1281/2560/2561 - Complete Datasheet*, San Jose, Atmel Corporation , 2014, pp. 236-264.

Appendix


Microcontroller
U1
RESET
XTAL2
XTAL1
AREF
AVCC
AGND
C4
0.1uF
VCC
GND
PL7
PL6
PL5(OC5C)
PL4(OC5B)
PL3(OC5A)
PL2(T5)
PL1(ICP5)
PL0(ICP4)
PJ7
PJ6(PCINT15)
PJ5(PCINT14)
PJ4(PCINT13)
PJ3(PCINT12)
PJ2(XCK3/PCINT11)
PJ1(TXD3/PCINT10)
PJ0(RXD3/PCINT9)
PH7(T4)
PH6(OC2B)
PH5(OC4C)
PH4(OC4B)
PH3(OC4A)
PH2(XCK2)
PH1(TXD2)
PH0(RXD2)
(AD7)PA7
(AD6)PA6
(AD5)PA5
(AD4)PA4
(AD3)PA3
(AD2)PA2
(AD1)PA1
(AD0)PA0
(OC0A/OC1C/PCINT7)PB7
(OC1B/PCINT6)PB6
(OC1A/PCINT5)PB5
(OC2A/PCINT4)PB4
(MISO/PCINT3)PB3
(MOSI/PCINT2)PB2
(SCK/PCINT1)PB1
(SS/PCINT0)PB0
(A15)PC7
(A14)PC6
(A13)PC5
(A12)PC4
(A11)PC3
(A10)PC2
(A9)PC1
(A8)PC0
(T0)PD7
(T1)PD6
(XCK1)PD5
(ICP1)PD4
(TXD1/INT3)PD3
(RXD1/INT2)PD2
(SDA/INT1)PD1
(SCL/INT0)PD0
(CLKO/ICP3/INT7)PE7
(T3/INT6)PE6
(OC3C/INT5)PE5
(OC3B/INT4)PE4
(OC3A/AIN1)PE3
(XCK0/AIN0)PE2
(TXD0)PE1
(RXD0/PCIN8)PE0
(ADC7/TDI)PF7
(ADC6/TDO)PF6
(ADC5/TMS)PF5
(ADC4/TCK)PF4
(ADC3)PF3
(ADC2)PF2
(ADC1)PF1
(ADC0)PF0
(OC0B)PG5
(TOSC1)PG4
(TOSC2)PG3
(ALE)PG2
(RD)PG1
(WR)PG0
ATMEGA2560AU
SOUND_A0
LCD_EN
LCD_RW
LCD_RS
FAN_PWM
FAN_ROT
FAN_SPD_DWN
FAN_SPD_UP
FAN_DRV2
FAN_DRV1
RTC_SDA
RTC_SCL
RTC_SQ
LCD_D7
LCD_D6
LCD_D5
LCD_D4
LCD_D3
LCD_D2
LCD_D1
LCD_D0
LCD
VDD
GND
K
A
DB7
DB6
DB5
DB4
DB3
DB2
DB1
DB0
E
RW
RS
VO
VDD
VSS
HELLO WORLD
HD44780 LCD
HD44780LCD-1602
X1
10k
VR1
TITLE: Lab3 Schematics
AUTHOR: Ellis Hobby
REV: 1
Date: 04/13/23
Sheet: 1/1


Motor Driver
IC2
FAN_PWM
FAN_DRV1
FAN_DRV2
1-2EN
VCC1
1A
4A
1Y
4Y
GND1
GND3
GND2
GND4
2Y
3Y
2A
3A
VCC2
3-4EN
L293D
DCMOTOR
VDD
VMOTOR
GND
Sound Sensor
SOUND_A0
VCC
A0
D0
GND
Sound Sensor
RTC
RTC_SCL
RTC_SDA
RTC_SQ
SCL
SDA
SQ
BAT
DS1307
BT1
Motor Control
R1
R2
R3
100k
S1
S2
S3
SPEED_UP
SPEED_DOWN
ROTATION
C1
C2
C3
0.1uF
FAN_SPD_UP
FAN_SPD_DWN
FAN_ROT
TITLE: Lab3 Schematics
AUTHOR: Ellis Hobby
REV: 1
Date: 04/13/23
Sheet: 1/2