Northeastern University
Michael J. and Ann Sherman Center for Engineering Entrepreneurship Education

Jolt

A Wearable Alarm Shock

November 15, 2018

Team

Thomas Keith
Seamus Winters
Michael Komatz
Michelle Stack
Nick Hulsey
Kayla Tannock
Piper Jacobs

Abstract

Jolt is a smart connected wearable with a unique alarm shock capability: instead of an audio alarm, this bracelet will reliably and safely pulse an electric shock through the wrist of the user to wake them up. Jolt offers safety and reliability that is beyond comparable devices. Reviews for competitor's products show a lack of consistency in the functionality of the alarm, a core requirement for an alarm design. Another unique aspect to our product is its advanced locking mechanism. For a shock alarm to work there needs to be contact with the skin, so if the bracelet fell off or could easily be taken off the user could miss the alarm and thus the product would fail to meet its core functionality. The printed circuit board (PCB) controlling the wearable houses an ATmega microcontroller (ATTiny84), a DS1305 real time clock, and a shocking circuit which communicate using a serial peripheral interface (SPI). The locking mechanism was prototyped using AutoDesk Fusion 360 with the final version being machined in steel, and the housing for the PCB is a flexible material 3D printed using FormLabs technology. The overall goal of the project was to utilize the product development cycle to create a smart connected wearable with reliable alarm shock and locking capabilities for our client.

I. Introduction

As a potential consumer product, Jolt intends to solve the common problem of oversleeping with an elegant solution: a quick mild shock. The aim of the project, as developed by a dedicated team of Generate engineers, is first and foremost to build the first work-like prototype for what is to become a more developed device with more numerous and robust features. At the end of the team's semester of development, a first prototype has been developed, consisting of two parallel hardware builds and two parallel electrical systems. The mechanical systems each consist of a unique, complex clasp mechanism. As well as an enclosure that supports and protects a custom printed circuit board (PCB) and other internal components. One electrical system uses a vibration motor, while the other has the electric shock as the primary alarm system. As the product advanced from its ideation stage to the creation stage, it has been the team's intention to provide a variety of viable solution paths, with refined and developed options. A rudimentary firmware has also been developed for the PCB, which is capable of communicating between the microcontroller and real-time clock in order to control the alarm function of the vibration motor. The major next steps for the product are to improve the user compatibility of the device by integrating the software with an interface such as a mobile application and streamlining the hardware design of the product.

II. Software Overview

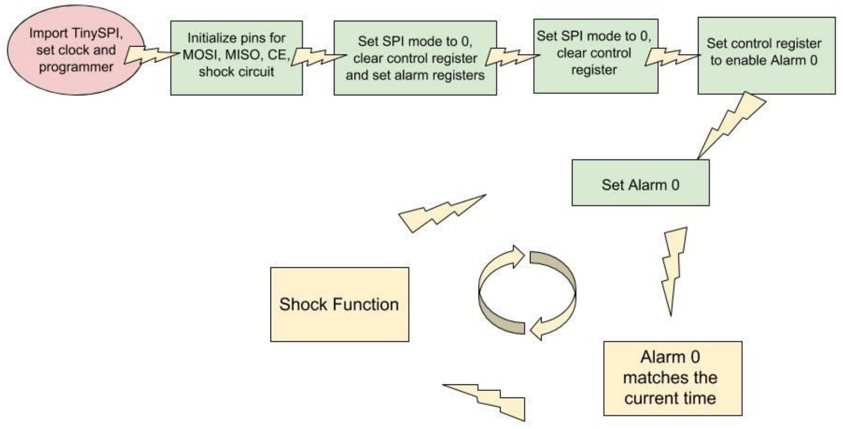
The software for this product is responsible for setting the alarm, as well as triggering the shock circuit when the alarm goes off. In order to do this, on board communication between the microcontroller and a clock must be developed such that it works reliably and consistently. Once communication is established, the software is not too complex, with only a few lines needed to set the alarm, and a constant loop to check when the alarm time is reached. Once the alarm time is achieved, activating the shock circuit is as simple as setting a specific pin on the board high to flip a switch within the shock circuit.

III. SPI

The general software problem to solve for this project was to develop on board communication between a real time clock (DS1305) and a microcontroller (ATTiny84) using a Serial Peripheral Interface (SPI) communication. SPI is a very common mode of communication used in microcontrollers with various applications; however, ATTiny microcontrollers do not support SPI, they support a Universal Serial Interface (USI), which can be programmed to act like SPI. In general, SPI works by sending data between a master and at least one slave. When the slave select line is high, that slave is in communication with the master. The master typically sends data over the MOSI line (Master Out Slave In), and the slave returns data over the MISO line (Master in Slave Out). However, because the Attiny84 does not directly support SPI, its data lines are flipped such that the MOSI line is actually the data out line for the slave, and the MISO line is the data in line. This means that for this communication, all data transfers from the MCU to the RTC occur over the MISO line, or data out line. Understanding this difference was the major turning point in the software development process. The main library used for this communication was TinySPI, written by Jack Christensen and open sourced on his GitHub. TinySPI was written for Arduino, so initially the Arduino IDE was used. This is an easy IDE to read, but it is very tedious to program in and does not fully support AVR-C editing, which we later realized we would need to dabble with. The final program used to develop the software was Atmel Studio, because of its ease of use and professional appearance. During the debugging process, the TinySPI library was rewritten in our program, and the main function used in our communication method, TinySPI.transfer(), was altered to be consistent with the specifications on our custom board.

TinySPI is a very efficient master library, but it is slightly limited as opposed to other SPI libraries that use various bit shifting techniques. Although TinySPI is only compatible with two of the four SPI modes, and does not support sending data least significant bit first, for our purposes it will work. TinySPI runs at one tenth the MCU system clock frequency, so for our board which runs at 16 MHz, TinySPI creates a bit clock with frequency 1.6 MHz, meaning that the microcontroller will take about 5 μs to transfer one byte to and from the RTC. The main software issue throughout this project was formatting the TinySPI library to our specifications, so to do this the TinySPI functions were added directly into our code so that they could be altered for our specific purposes. The final product of the code essentially contains a SPI library based off TinySPI.

IV. Software Flow

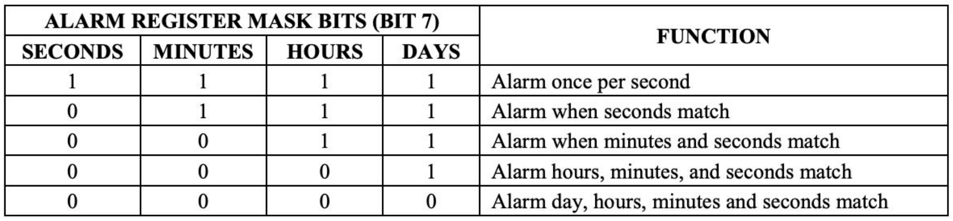

Import TinySPI, set clock and programmer
Initialize pins for MOSI, MISO, CE, shock circuit
Set SPI mode to 0, clear control register and set alarm registers
Set SPI mode to 0, clear control register
Set control register to enable Alarm 0
Set Alarm 0
Alarm 0 matches the current time
Shock Function

The flow of the software for this project is mostly setup, followed by a fairly simple loop that deals with setting and checking the alarm. The first step is an initialization step to make sure the necessary libraries are in the main file. The SPI functions (lines 170-222) were changed from TinySPI functions to perform the specific bit shifts at our specific pins. Setting the clock and programmer simply consist of setting the external clock frequency to 16 MHz for our board (line 36), and connecting the TinyISP programmer that we used. The first block of code (lines 34 - 56) imports other necessary libraries and defines any variables used throughout the program. These variables consist of pin numbers for the SPI lines, data lines and the motor/shock pin. Once these variables have been defined and the necessary libraries are included, the main function runs. The first step in programming this board is to call the begin function, which initializes the SPI lines as inputs or outputs on the MCU so that data transmission can occur. TinySPI was written to function in SPI mode 0 or 1, and as long as we stay consistent with one of them it doesn't matter too much which we used. The SPI data mode being used is data mode 1, which means that clock polarity is 0 which sets the clock to idle at 0 (rising edge is the leading edge), and the clock phase is 1 which means each bit transmission will occur halfway between the rising and falling edges of the clock cycle.

The next step is to reset the internal clock time on the RTC with our first data transfer by writing 0x00 to the clock's seconds register. The transfer function used to send data must occur after the chip enable line is driven high. This pin number is stored in the variable CE_PIN, and a flip latch function was written to take an input of 0 or 1 and based on the input drive the chip enable pin high or low. This function is used before any data transfer to drive the pin high, and again after the transfer to drive it low. The data transfer function takes an input byte to be sent from the MCU to the RTC, and is always called in groups of two. The first byte transferred is the data register for the second byte transferred to be stored at. When reading from the RTC, the second call to the transfer function returns the value stored at the register whose address was transferred first. An example of reading from the RTC can be found on lines 128 - 131, when the status register is read. The next step in the program is to clear the control register on the RTC. The byte stored in the control register can cause the clock to automatically perform some operations, so clearing it is necessary because any lingering data in the control register could cause inconsistent functionality. The byte we want to write to the control register was derived from page 7 of the DS1305 data sheet, where the individual purpose of each bit in the control register is explained. Writing 0x05 to the control register will activate the Alarm 0 interrupt pin as well as activating the flag bit in the status register, which gives us two ways to check if the alarm has been triggered.

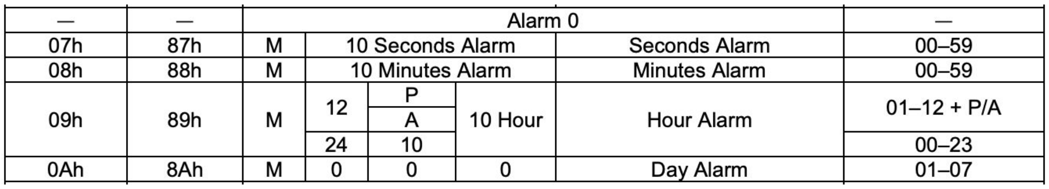
After writing the control register, the next step is to write to the alarm 0 register in order to set our alarm. The registers are easy to find based on the data sheet, but deriving the bytes to write to those registers proved to be fairly complicated. The data sheet provides two charts for decoding what bytes to write to which registers. The first important concept is the mask bits of each register. Table 2 from the data sheet shows how mask bits (also known as bit 7 in each register) determine alarm frequency.

Table 2. TIME-OF-DAY ALARM MASK BITS



ALARM REGISTER MASK BITS (BIT 7)				FUNCTION
SECONDS	MINUTES	HOURS	DAYS	
1	1	1	1	Alarm once per second
0	1	1	1	Alarm when seconds match
0	0	1	1	Alarm when minutes and seconds match
0	0	0	1	Alarm hours, minutes, and seconds match
0	0	0	0	Alarm day, hours, minutes and seconds match

For our purposes, we want to generate an alarm dependant on seconds, so that when the seconds written to the alarm register match the seconds in the clock register, the alarm is triggered. This means that the byte in the seconds register starts with a 0, and the bytes in the minutes, hours, and days registers all start with a 1. If an alarm was meant to go off at a certain minute, such as 5:30 AM, then the mask bits for the seconds and minutes registers should be 0. The next step is to figure out the next 7 bits of each byte. To do this we used the Alarm 0 registers from Figure 2 on the data sheet.



—	—	Alarm 0					—
07h	87h	M	10 Seconds Alarm			Seconds Alarm	00–59
08h	88h	M	10 Minutes Alarm			Minutes Alarm	00–59
09h	89h	M	12	P / A	10 Hour	Hour Alarm	01–12 + P/A
			24	10			00–23
0Ah	8Ah	M	0	0	0	Day Alarm	01–07

For our alarm, we want to create an alarm that generates repeatedly on an interval of x amount of seconds, so we store the alarm frequency in a variable in seconds. Then, we write this value to the seconds register, 87h, with a mask bit of 0. The minutes, hours, and days alarm should all be zero, so writing all 0's with a mask but of 1 results in the hex value 0x80 (10000000 in binary). The method of setting these registers is to store the register values in one array, and the bytes to write in another. This makes it easy to loop through the arrays writing bytes to their corresponding registers (lines 117 - 122). With this format for setting the alarms, the only values that should change for different alarm values are the values in the alarm bytes array (line 90).

Once the alarm has been set and the control register turned on the alarm functionality, the program loops until the alarm and clock values match. To detect this, the program repeatedly reads from the status register using the transfer methods, and stores the status register byte in a variable called grab (lines 128 - 131). Based on the status register in the data sheet, when alarm 0 goes off, bit 0 of the register will be set to a 1.

STATUS REGISTER (READ 10h)

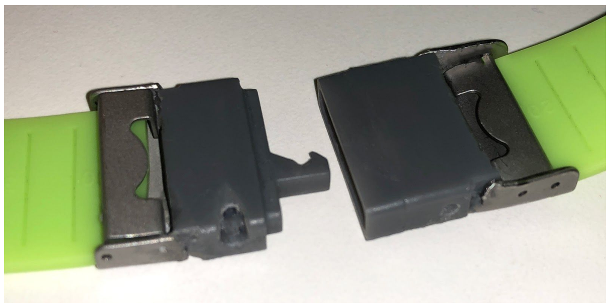
BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0	0	0	0	0	0	IRQF1	IRQF0

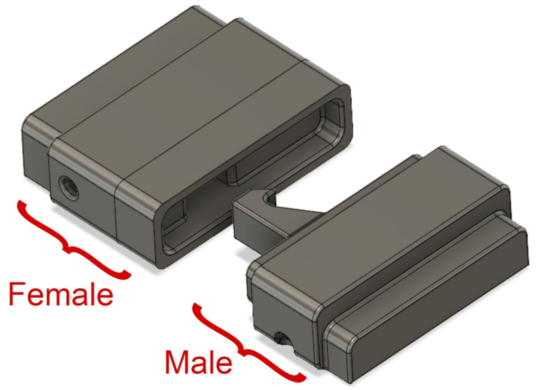
This is helpful because it allows us to check the status register byte, and as soon as it changes to 0x01, activate the shock circuit. Once the alarm goes off, the flag bit in the status register has to be cleared, which automatically happens after any read or write to alarm 0 (lines 138 - 141). Also, the internal clock is reset for our purposes (lines 144 - 147), which allows the alarm to act as a timer where it repeats on a certain interval. If the alarm is to be used as a time of day alarm, this section should be commented out.

The final section of our program is the pulse generating function. The shock circuit is connected to a pin on the MCU so that by writing this pin high and low for certain intervals, the circuit can send a shock pulse to the user. The pulse function (lines 159 - 168) loops based on the variable PULSE_AMOUNT such that if this variable is set to 2, the function generates 2 shocks. The time the shock is on is determined by PULSE_TIME in milliseconds, and the time between shocks is determined by PAUSE_TIME in milliseconds. This method of generating a pulse makes it so the entire shock wave is programmable from three variables to the specifications of the user.

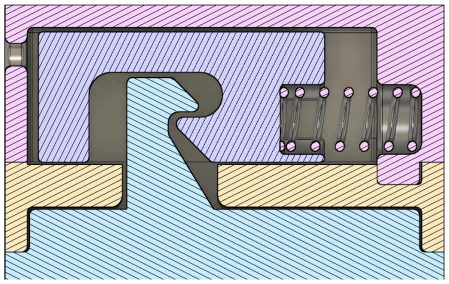
V. Hardware Prototype 1

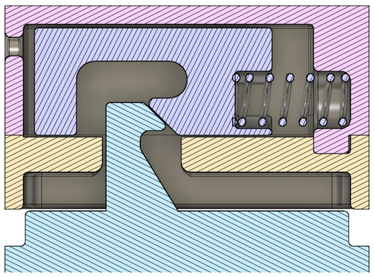
The three main hardware components of the product are the locking mechanism, the protective housing, and the flexible strap. Because the locking mechanism was such a unique feature, this is where most of the mechanical development took place. The two parallel prototypes accomplish the client's desire for adjustability, lockability, and sleek design. The current stage of the mechanical assembly is two working prototypes that provide a sound starting point for future development. The first prototype, designed by Michael Komatz, implements a spring lock design which consists of five components - the male, female body, female cap, pusher, and spring. Additionally, a paper-clip sized pin is used with the device and can be stored in the male component. All parts except for the spring and pin were printed in SLA using a Form 2 3D printer.





The male part of the lock consists of a hook feature that interfaces with the pusher inside the female body to allow for locking capabilities. This component is intentionally stepped to allow for a secure and flush enclosure when inserted into the female part. The hole in the side of the male is a dock for the pin used to unlock the device. The use of a small pin to unlock the device requires the user to reach a certain level of consciousness to detach the wearable, which will hopefully decrease the likelihood of the user pulling off the wearable and falling back asleep, which was a major concern of the client's. The female part is composed of a body, cap and pusher, which work in unison to apply force to the spring as the two parts are connected, and then as relief for the spring once the lock is secured.



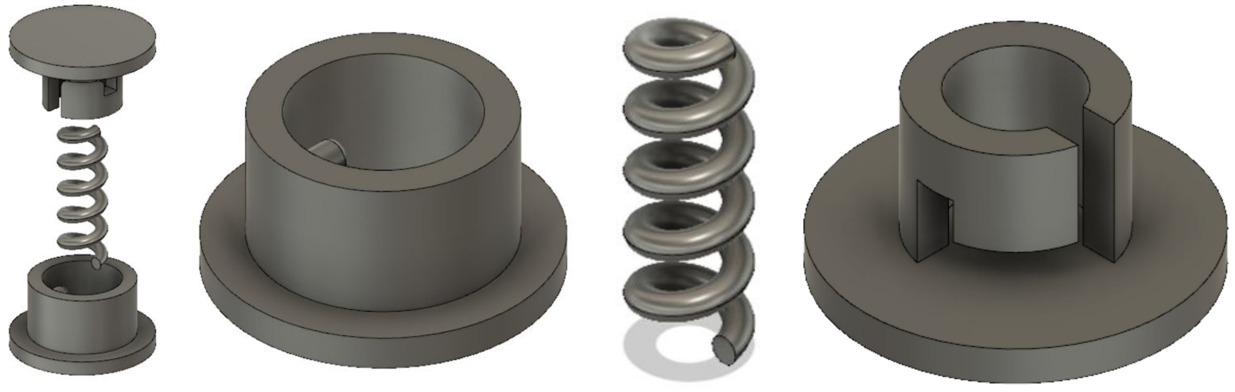


The next hardware component is the housing for the electrical components, which was also printed from the Form 2 in SLA material. The housing was built to accommodate the first revision PCB and provide a means to equip the device to a user's wrist. The primary goal was to minimize its overall footprint while providing enough support for use while testing. Its main features are slots where the ends of the straps were glued into, as well as holes for ports for the battery plug and microUSB. Additionally, the housing had small supports in the inner corners that the PCB would rest on, which allowed enough space for the coin cell battery holder to fit.

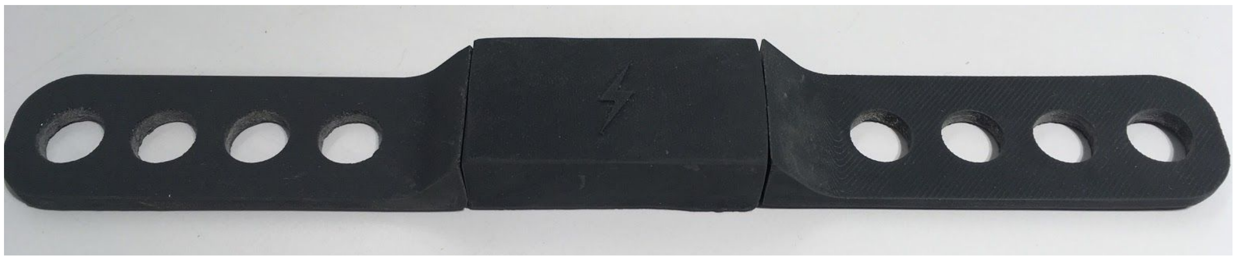
The final mechanical subsystem was the strap for the wearable, which was repurposed from a Road I.D. Bracelet. The ends of the clasps on the Road I.D. bracelet have sharp "clamps" that grab onto the silicone, keeping it attached. This type of mechanism would not have been effective using 3D printing materials because of its lack of strength; however, we were able to cut and superglue these clamps onto either end of the locking mechanism for a demonstration of how combining that function with our spring lock design might work.

VI. Hardware Prototype 2

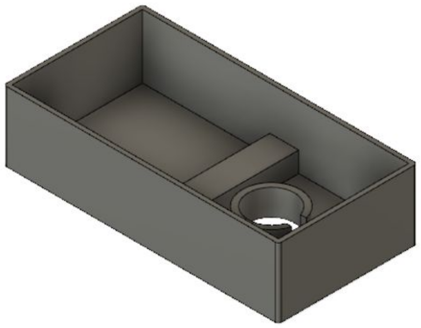
The second prototype for the hardware components, designed by Piper Jacobs, consists of an SLA band, a unique housing for the electrical components, and another spring lock mechanism. These designs were more of look-like designs than work-like designs due to time constraints on streamlining the size of the electrical components. The spring lock mechanism is similar to the first prototype because the user is required to apply force to the spring in both cases, but this design also requires the user to twist the locking mechanism while applying force to it. While not too complicated to unlock, the twisting action required by the design requires the same level of consciousness to achieve with one hand, which would again fulfill the client's wishes for a complex locking mechanism.

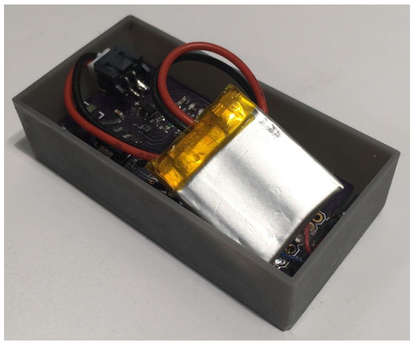


The strap design to go with this locking mechanism would require the lock to be on opposite sides of the strap, and be inserted to the desired strap hole, similar to the way a common belt lock works. The variant hole locations achieve the adjustability of the strap in a more temporary way that the first prototype, which requires the user to cut the strap to size. The strap was printed in SLA from the Form 2, and was designed to hold the housing within it rather than be fed through the housing as in the first prototype.



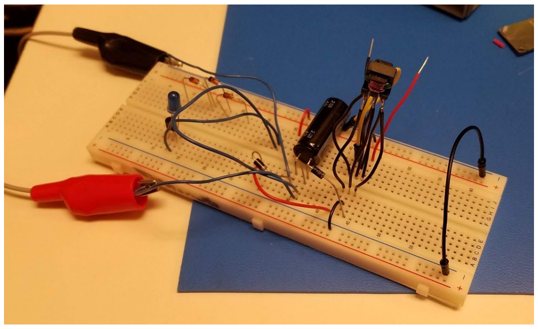
The housing for the second prototype contrasts the first prototype mainly in that it allows direct contact between the skin and the shocking mechanism, which would allow for a more consistent shock to the user.

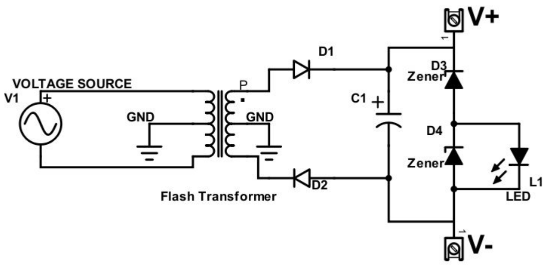




VII. Electrical Overview

The two main electrical subsystems developed for this product were the custom PCB and the shocking circuit. The main components of the PCB are the microcontroller and the real time clock, as well as a lithium polymer battery charging circuit that powers the PCB. The shocking circuit is currently a breakout board for the custom PCB. The circuit was modeled after flash circuitry of cameras, as they operate at a similar output voltage required to sustain a reasonable, yet safe shock. The determined safety parameters were taken from multiple studies, including the International Electrotechnical Commission's report and standardization of electric current, body impedance, and the body's voltage thresholds. With these standards in mind, the output current was regulated at 0.011A, which is well below the safety requirements of a shocking implement. The microcontroller controls the switch for the circuit, as well as the AC source for the circuit's oscillator. The circuit's configuration consists of a 6 pin flash transformer oriented in reverse in order to perform step up operations. The center pins on both sides are grounded. On the secondary side of the transformer, 2 diodes are connected in parallel with a 9μF capacitor and the probes. The configurable section of this circuit is between these probes. Connecting a selection diodes in series with each other, but in parallel with the probes, allows the output voltage to be controlled by these diodes, while the LED in parallel with the diodes will signal when that voltage has been reached. Although the shock circuit is only implemented in a work-like breadboard prototype, the functionality was confirmed. The next step for the electrical components is to implement the shock circuit on a next iteration of the PCB, as well as streamlining the PCB to be more efficient and concise to allow for sleeker external design.





VIII. Next Steps and Future Compatibility

The next steps for the product as a whole are to connect it with a user interface allowing the alarm to be set and reset by a user, as well as streamlining the design of the product to make it more visually appealing. The main goal for the electrical subsystems is to create a second iteration of the PCB with the shocking circuit included, while at the same time shrinking the current PCB specifications. Once a new PCB is made, the two mechanical prototypes could be tested and potentially combined to create one final prototype that maintains the functionality of the first prototype while adding the adjustability and sleek design of the second prototype. A main goal for this project, especially the software side of it, was to develop the product in such a way that it was compatible with future interfacing for the client. The final version of the software for this device has 7 variables that can be changed to access its entire range of functionality. In the next prototype of the product, an interface could be created where a user enters values to be passed into the current program. The only necessary values would be a time of day for the alarm, and if the user should have control over the size of the shock, three values to control that: the time on and time off of each pulse, and the number of pulses desired. The interface could even request a level of intensity from the user, and within the program determine those pulse values from the given intensity. By reducing the amount of editable code within the program, the software is well positioned to interface with an app or some other form of control, specifically Bluetooth. A reach deliverable for the project was Bluetooth compatibility, so although there was not sufficient time to achieve this, setting it up to be compatible with Bluetooth in the future was a major step in the process.

IX.

X. Software Example

The alarm time can be set from one array of bytes within the code, with four variables changing to generate any time of day alarm. For example, if the alarm is to generate every day at 5:30 AM in 12 hour mode, the four bytes in alarm bytes should be: 0x80, 0x45, 0x30, 0x00. The first byte has mask bit 1, and since it is the days register this will generate an alarm whenever the hours, minutes and seconds match the clock, or once every day. The next byte is the hours byte, and will corresponds to 01000101 in binary. Bit 6 being high activates 12 hour mode, and bit 5 being low sets it to am. If the desired time is 5:30, then the hour is 5, or 00101 in binary. The minutes byte is set to 0x30 for 30 minutes, and the seconds byte is set to 0 for 0 seconds. This results in a time of day alarm being flagged every time the clock's internal time matches 05:30:00 AM. Keep in mind, the clock reset snippet should be commented out (lines 96-99), or else this will generate an alarm every five and a half hours. This alarm would generate our current pulse, but if that was to be changed as well, the three variables corresponding to the waveform could be changed.