High Current Pulse Generator for the Application of Transcranial Magnetic Stimulation

Team Number:	SDDEC18-04
Client:	Dr. Mani Mina
Advisors:	Dr. Mani Mina Dr. Jayaprakash Selvaraj Dr. Priyam Rastogi Dr. Neelam Prabhu Gaunkar Tom Kimler Wei Shen Theh
Team Members:	Abdullah Bahashwan Brian Kirkpatrick Curtis Richards Jonathan Rothfus Tania Alvarado Carias Yan Wang
Team Email:	sddec18-04@iastate.edu
Team Website:	http://sddec18-04.sd.ece.iastate.edu
Revised:	12/14/18-V1

Table of Contents

1 Introductory Material

1.1 Acknowledgement

We would like to acknowledge Iowa State University for its financial assistance. This project would not be possible without their support. We would also like to acknowledge the valuable technical assistance and guidance of the following individuals:

Dr. Mani Mina

Dr. Jayaprakash Selvaraj

Dr. Priyam Rastogi

Neelam Prabhu Gaunkar

Tom Kimler

Thank you!

1.2 Problem Statement

This project aims to develop an affordable high current pulse generator for the application of Transcranial Magnetic Stimulation (TMS). This technology is used to generate a pulsing magnetic field that can be focused on regions of the brain to treat various brain disorders. TMS has been studied for treatment of brain disorders since the mid 1980's and has been approved for the treatment of depression in the US since 2009 [1]. It is currently being researched for treatment of other disorders including schizophrenia and Parkinson's.

Currently, there is a need for a more cost effective and customizable pulse generator for use in basic research. Existing approved commercial TMS units are very expensive and cannot be used with coils other than proprietary ones provided by the manufacturer. In recent years, different types of coils have been developed [2]. Moreover, different TMS coil have a tradeoff between focality and the depth of penetration [3]. By building and improving on past research and experimentation in this area, combined with innovations and original design, our team is aiming to meet this need for a robust, lower cost and flexible device that can be used with a variety of coils in a research setting.

This project will help to advance research in an area that has great social significance. Depression and other mental disorders are often treated with medication that causes dependency, is expensive, and has many negative side effects. TMS has already shown great promise in treating depression non-invasively and without medication. Additionally, the results of the treatment seem to be persistent, meaning the patient may not need to continue treatments after a few sessions. This technology shows promise for improving lives in a very real way, and we are eager to contribute.

1.3 Operating Environment

The intended operating environment for the final TMS device is a controlled one, due to its planned use in a research setting.

We do not anticipate nor design for this device to: be exposed to extremes in temperature, humidity, pressure, or particulates. The device is however likely to be used for extended periods of time, by multiple users, and in multiple physical locations (primarily laboratories). As a result, the device should be portable and robust enough to be easily moved from one location to another by 1 or 2 people without damaging the device or injuring the users.

The device has been designed to operate from a standard 120 Volt 60 hertz wall outlet and will therefore be exposed to associated surge or power outage risks.

1.4 Intended Users and Intended Uses

Our primary intended users are researchers familiar with TMS technology working in a lab setting. Qualified students will likely also have access to the device under supervision or with training. The device is intended to be used under the control of a GUI to generate high-current pulses and vary the parameters of those pulses - pulse width, amplitude, total duration of operation.

A typical use case would involve a researcher attaching a compatible coil to the TMS device dependent on the size and shape of magnetic field needed for the research being done and test subject anatomy. The user will plug the device into a wall outlet, connect the device to a computer running Matlab and open the Matlab GUI provided. Once the GUI is up, the device is connected to the wall and the appropriate coil attached the user can operate the device via the GUI.

1.5 Assumptions and Limitations

Our TMS device is intended to be used under the following **limitations**:

- The total cost to produce the device shall not exceed $3000 dollars.
- The device shall support 1 coil type at a time. Multiple coils shall not be used simultaneously.
- The device shall <u>not</u> be used for TMS research on human subjects unless future approval is received.

Our TMS device was designed and made under the following **assumptions**:

- The device shall be used with US standard 120 V, 50-60 Hz wall current.
- The device shall be used only by or under the supervision of trained operators.
- The device shall be capable of sending repetitively 10 pulses a minute.
- The device shall be capable of varying all pulse parameters described in section 2.2 Functional Requirements via the provided GUI.

2 Proposed Approach and Statement of Work

2.1 Objective of the Task

Our team is to design, build, and test a high current pulse generator for the use of transcranial magnetic stimulation coil testing.

2.2 Functional Requirements

The device shall be designed to generate a pulse of current with an amplitude up to 2,000 Amperes. The pulse width shall be able to be modulated between 400 – 27 microseconds. It shall be able to produce up to 10 pulses per minute. All the previously mentioned features shall be controlled using a GUI.

2.3 Constraints & Considerations

The team shall create a graphical user interface to input the desired outputs from the machine. Standard IEEE code writing protocols shall be followed throughout the project.

2.4 Previous Work and Literature

In the past, other teams have developed and built plans for a TMS high pulse current generator. Their machine was designed to reach a peak current of 1000 Amperes, a pulse width between 50-400 microseconds, have an easy to use GUI, and a budget of $500 [4,5]. Our own project's objectives are mentioned earlier in section 2.2.

As our team researched past projects, we found the criteria that the teams had the most trouble meeting was cost and the size constraints of components causing safety concerns [4,5]. We are combating this by adding in gate checking throughout our circuit design phase.

In industry, there are several commercial options available. Magstim is the one our team is most familiar with. This machine can reach all our own objectives, however there is a reason we have been tasked with it. The shortcomings of using this machine in research are the high cost and difficulty of interchanging homemade coils. While our own circuit design will have very similar componentry and design as past projects, our project shall be able to reach higher current capability than past Iowa State projects and have an easier platform than Magstim to interchange various coil designs.

2.5 Safety Considerations

Due to the amount of current that is going through the circuit, safety measures need to be taken. After reviewing past projects, we've seen how some of them have open cables freely hanging outside the black box, which is not safe as contact with a wire when the circuit is running is dangerous. The only wires that would be outside of the box are the wall plug and the connections to the coil.

During testing we used alligator clips which proved to be not strong enough for high current. Now that we finalized testing we changed the connections with terminal buses which will be more efficient. The plan for insulating the buses and the coil is to cover it with heat shrink potentially the coil with electrical tape. Whenever the coil is changed, the heat shrink will have to be replaced. This way there won't be any copper loosely hanging and no one using the machine could get hurt.

The user of the box will have easy access to the emergency discharge button placed on the wall of the box. With this, if they were to open the box or run the circuit after having ran it earlier they are able to make sure that the capacitors are fully discharged. The charge of the capacitors will be easy to read using the analog voltmeter attached to the side of the box. The resistors used to discharge the capacitors are very strong and are designed to easily discharge the maximum voltage that the capacitors can hold.

3 Project Design

3.1 Proposed Design

Delivering 2000 Amperes through a 12 american wire gauge (AWG) coil for only 400 microseconds is a large task to undertake when designing for vast magnitudes of current at high speeds. Past teams have taken on similar TMS projects dealing with much less current. A discussion of their designs and trials can be found in 5.3 Appendix II.

The original designs for this teams project were based off of ideas found in Polson's Patent, Magnetic Stimulator [6]. This design incorporated high power thyristors and allowed for the energy released in the coil to be regenerated into different capacitor banks. An analysis of this can be found in 5.3 Appendix II. Unfortunately due to time, money, and the lack of component restraints, this design had to be abandoned and a near new model developed in what was a rather short time period.

There was only two weeks between when the thyristor design was abandoned and when parts needed to be ordered to keep our project on schedule. Taking into great consideration to our roots in circuit theory, Kirchhoff's Current Law: All currents entering a node must be equal to the currents leaving the node, became the design law to reduce cost while reaching our current levels with available components. As seen in figure 1, the design takes advantage of different modules or legs of capacitors and insulated gate bipolar transistor (IGBT) to supply large amounts of current that none of the components could safely handle on their own. By using IGBTs we are able to closely control the waveform shape and magnitude of the current seen by the coil. This was inspired from the mouse pulsar design [7].

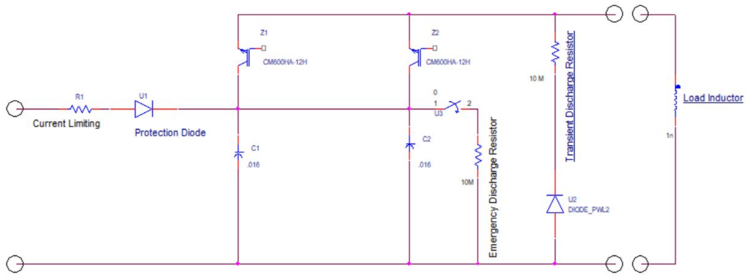


Fig. 1 Power Circuit Design

To dissipate the current in the load, we introduce a diode and resistor pair in reverse polarity across the circuit. While delivering the pulse our diode is non-conducting, but when the IGBT is turned off, the load reverses polarity and turns our dissipation diode into a conducting path for our load current to move through the resistor and transform into heat.

The IGBT was controlled through a microcontroller. Originally the team used a standard Arduino Uno, but then switched to the IEIK brand ATmega328 shown in figure 2. This is an exact clone of the popular Arduino UNO available for less than 50% of the cost of the original Arduino. This microcontroller is embedded in the TMS device and powered by a standard 9V, 2A “wall wart” power supply shown in figure 3.

Fig. 2 The Arduino Clone

Fig. 3 Wall Wart Power Supply

The following Absolute Maximum Ratings for the microcontroller are relevant:

40 mA max DC current sink or source per IO pin,
20 V max DC input per IO pin,
20 V max DC power supply input.

This microcontroller’s IO pins are limited to sinking or sourcing a low current. To prevent overcurrent, current limiting resistors were used on all IO pins connected to the TMS device and current was tested using the multimeter shown in 5.4 Appendix III.

The circuit expectations were all met with the latest design, but it was not feasible to include the regenerative circuit in the redesign due to time constraints. Other design features were added in with testing. Those being a bluetooth control module for our microcontroller, so no computers can be directly connected to the machine. A digital voltage regulator was also added so that the user can choose to charge the capacitor banks to 50% capacity to decrease wasting electricity depending on the current need.

3.2 Design Analysis

One of the strengths of the design is the “modularity”. Each capacitor/IGBT line is equal to one resistor/diode dissipation line, both connected at the same node. As one can read in the coming section on testing, we began testing one capacitor/IGBT line before adding additional lines. The success of this found in testing proves that our peak current magnitude is limited only by cost and space, not the design.

3.3 Design Simulations

Throughout the two semesters of the project, simulation softwares were used frequently. At the beginning, simulations were used to test our initial design both as a whole or smaller parts. The IGBT used was simulated with the coils provided many times to ensure proper functionality before testing the actual design built.

Figure 3 below shows the characteristics of the IGBT used obtained by PSpice and figure 4 shows the current form simulating different coils with changing inductance.

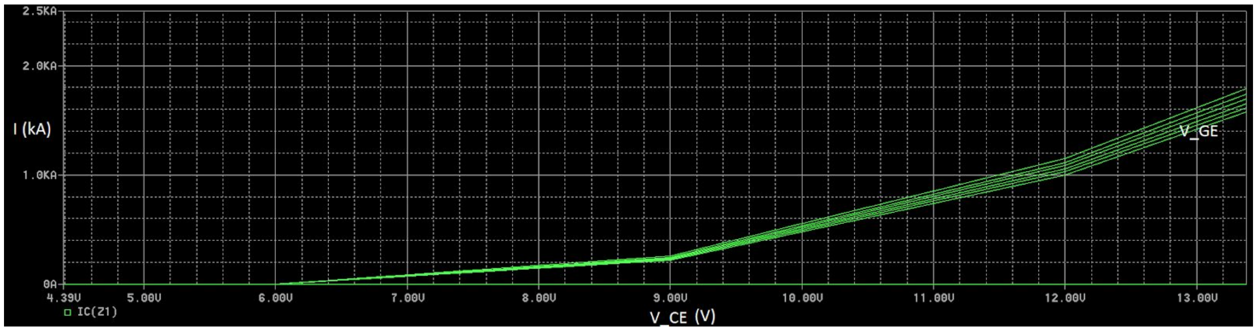


Fig. 3 IGBT Characteristics in PSpice.

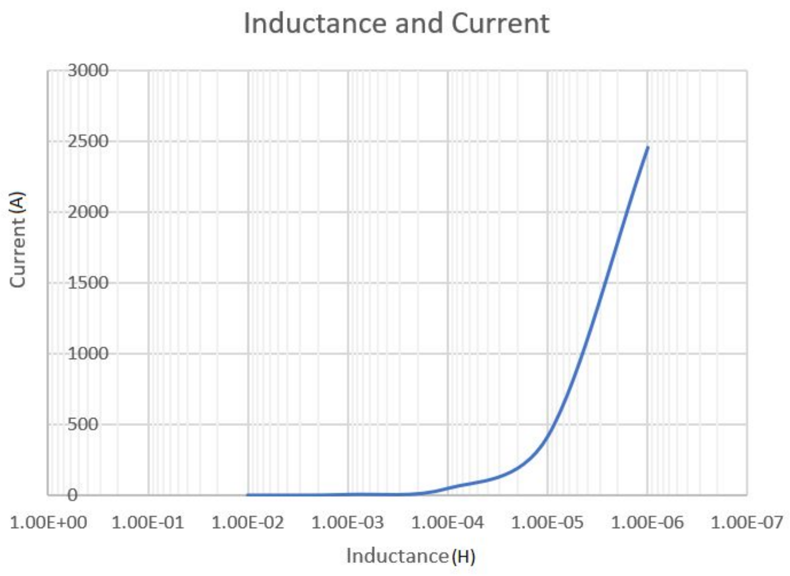


Fig. 4 Current Vs. Inductance.

4 Testing and Implementation

4.1 Interface Specifications

4.2 Hardware

4.21 Power Circuit Testing

We started the Power Circuit Testing with testing the IGBT. These tests are based off IEEE Standards [8]. Initially we passed a DC signal through it with a pulse on the gate. Later we implemented one of the IGBT's into the circuit and connected it to one of the capacitors. This way we used the signal from the capacitors as an input and continued using a function generator for the gate. Once we proved that to be successful we finished implementing the rest of the circuit, connecting the second IGBT and second capacitor to the circuit and continued using the function generator as a gate signal. In the results section we discuss the output of the testing in more detail. Our final design uses the microcontroller instead of the function generator for the gate signal.

During testing we experienced some difficulties with the IGBT's. We tested them to see if they were damaged by checking if there was a short through them. One of the IGBT's came in faulty and another was damaged throughout testing. Due to this and knowing these components are quite expensive we made sure to be very careful throughout their use in testing.

Through rather meticulous testing, it is shown through the use of a gauss meter as well as oscilloscope that we can get a max output currents in the 320-350 amps per IGBT consistently with the coils given. Due to the inability to accurately measuring resistance of said coil, we chose instead to use the gaussmeter to measure the magnetic field through the coil and with that bit of information we could calculate current using the long solenoid approximation method ($B=\mu nI$). To further the integrity of our findings, we decided to incorporated a hall sensor as a mode of measurement for the currents through the coil. The math was as followed with the coil.

In this coil's case, the length was 7.5 centimeter, with 16 turns. We used the equations $B=\mu nl$, $\mu=k\mu_0$ and $n=N/L$. The givens were $\mu_0 = 4\pi*10^{(-7)}$ H/m and $k = 1$ given that the coil was made of copper. In our case $\mu=\mu_0$ and $n = 16/(.075)$. Thus we are now only missing two unknowns which is the current(I) and the magnetic field(B). Thus by using the gaussmeter which is used to detect magnetic fields, we can eliminate the final unknown of I through calculations.

4.22 Rectification Circuit Testing

The rectification circuit is a full bridge rectifier that takes our transformer voltage and outputs a 240 volt DC. This was designed and tested using Multisim, Falstad and Eagle. We used a series of 1N4006 diodes to have a diode that could handle high voltage. We implemented fuses as an additional means of protection. When testing the capacitors charging we monitored the current to determine the rating of the fuses.

4.23 Electronic Measurement Testing

Implementing a measuring circuit to monitor the charge on the capacitor with the microcontroller when the voltages sweep from 0-240+ Volts. On our first design we used implemented an LED block that would activate in 60 volt intervals, effectively 25% increments. When testing this we realized the increments were a bit limiting and we changed the design to allow a continuous reading on the microcontroller. Through some simple programming we were able to keep our charge indicator percentages displayed through the GUI.

4.3 Microcontroller Testing

4.31 Matlab Graphical User Interface (GUI) Testing

The GUI was tested for functional and non-functional requirements. Functional requirements are those given specifically by the Client which the device must meet, such as the ability to control the number of pulses and the frequency of the pules. Non-functional requirements are requirements which aim to make the GUI robust, user-friendly, scalable, etc.

On the Microcontroller side, the Arduino Serial Library's Serial.println() method was used extensively to ensure that the correct command characters and number of bytes were received by the Microcontroller from the GUI. Each valid command character that could be received by the MC was verified with the series of print statements as in the code snippet example below:

```
case CHANGE_FREQUENCY:
        Serial.println("In frequency case. Command is:");
                        Serial.println(command);
        set_frequency();
        Serial.println("Frequency set to:");
        Serial.println(frequency);
        Serial.println("Bytes available:");
        Serial.println(BT_serial.available());
                                    break;

          case CHANGE_PULSE_COUNT:
        Serial.println("In count case. Command is:");
        Serial.println(command);
                        set_pulse_count();
        Serial.println("pulse_count set to:");
        Serial.println(pulse_count);
        Serial.println("Bytes available:");
        Serial.println(BT_serial.available());
                                    break;
```

The following test cases cover each functional discrete GUI item, such as a button or a knob. In the test cases described below the term "the Application" means the Matlab code that is running the GUI and interacting with the Microcontroller. The term "Microcontroller" describes the embedded Arduino board described above, plus the connected Bluetooth module that is receiving commands from the GUI.

"Disconnect Microcontroller" Button Testing

Test Case 1

Expected Results: When a user clicks on the "Disconnect Microcontroller" button and the Application is already disconnected from the microcontroller, the Application shall present the user with a dialogue box confirming that the microcontroller is disconnected.
Inputs/State: Application is disconnected from the Microcontroller and a user clicks the "Disconnect Microcontroller" button.
Results: Test Passed.

Test Case 2

Expected Results: When a user clicks the "Disconnect Microcontroller" button and the Application is connected to the microcontroller, the Application shall disconnect from the Microcontroller serial object and present the user with a dialogue box confirming the disconnection.
Inputs/State: The Application is connected to the Microcontroller and a user clicks the "Disconnect Microcontroller" button.
Result: Test Passed.

"Connect Microcontroller" Button Testing

Test Case 1

Expected Results:

When a user clicks the "Connect Microcontroller" button and the Application is not connected to the Microcontroller, the Application shall connected with the Microcontroller serial object and present the user with a dialogue box confirming the established connection.

Inputs/State: The Application is not connected to the Microcontroller and a user clicks the "Connect Microcontroller" button.

Results: Test Passed.

Test Case 2

Expected Results:

When a user clicks the "Connect Microcontroller" button and the application is already connected to the Microcontroller, the Application shall present the user with a dialogue box confirming that the Application is connected to the Microcontroller.

Inputs/State: The Application is connected to the Microcontroller and a user clicks the "Disconnect Microcontroller" button.

Results: Test Passed.

"Pulse" Button Testing

Test Case 1

Expected Results: When the user clicks the "Pulse" button and the Capacitor Charge Level is less than Power Level 1 the Application shall not send a "pulse" command to the Microcontroller serial object. The Application shall present the user with a dialogue box prompting to charge the capacitors and try again.

Inputs/State: The Application is connected to the Microcontroller. The Capacitor Charge Level is less than Power Level 1. The user clicks the "Pulse" button.

Results: Test Failed.

Test Case 2

Expected Results: When the user clicks the "Pulse" button and the Application is not connected to the microcontroller, the Application shall not send the pulse command to the Microcontroller serial object. The Application shall present a dialogue box to user prompting to connect to the microcontroller and try again.

Input/State: The Application is not connected to the Microcontroller and a user clicks on the "Pulse" button.

Results: Test Passed.

Test Case 3
<u>Expected Results:</u> When the user clicks the "Pulse" button and the Capacitor Charge Level is Power Level 1 or greater, and the Application is connected to the Microcontroller, the Application shall send the "pulse" command to the Microcontroller serial object.
<u>Inputs/State:</u> The Application is connected to the Microcontroller. The Capacitors Charge Level is Power Level 1 or greater. The user clicks the "Pulse" button.
<u>Results:</u> Test Passed.

"Frequency" Drop Down Menu Test Cases
Test Case 1
<u>Expected Results:</u> When the Application is not connected to the Microcontroller and the user attempts to change the Frequency, the Application shall not send the selected Frequency to the Microcontroller serial object. The Application shall reset the Frequency Drop Down Menu to the previous value and present the user with a dialogue box prompting to connect to the Microcontroller and try again.
<u>Inputs/State:</u> The Microcontroller is disconnected. A user selects a Frequency from the Frequency Drop Down Menu.
<u>Results:</u> Test Passed.

Test Case 2
<u>Expected Results</u>: When the Application is connected to the Microcontroller and a user selects a Frequency from the Frequency Drop Down Menu, the Application shall send the selected Frequency to the Microcontroller serial object. The Application shall then present the user with a dialogue box confirming that the Frequency has been set to the selected value.
<u>Inputs/State:</u> The Microcontroller is connected to the Application. A user selects a Frequency from the Frequency Drop Down Menu.
<u>Test Results:</u> Test Passed.

Test Case 3
<u>Expected Results:</u> When the Application is first launched or restarted, the Frequency Drop Down Menu setting shall be set to its default value as defined by the macro #define STARTUP_FREQUENCY 1
<u>Inputs/State:</u> The Application is launched or restarted.
<u>Results:</u> Test Passed.

"Pulse Count" Drop Down Menu Testing
Test Case 1

Expected Results: When the Application is not connected to the Microcontroller and the user attempts to change the Pulse Count, the Application shall not send the selected Pulse Count to the Microcontroller serial object. The Application shall reset the Pulse Count Drop Down Menu to the previous value and present the user with a dialogue box prompting to connect to the Microcontroller and try again.
Inputs/State: The Microcontroller is disconnected. A user selects a Pulse Count from the Pulse Count Drop Down Menu.
Results: Test Passed.

Test Case 2
Expected Results: When the Application is connected to the Microcontroller and the user changes the Pulse Count, the Application shall send the selected Pulse Count to the Microcontroller serial object. The Application shall then present the user with a dialogue box confirming that the Pulse Count has been set to the selected value.
Inputs/State: The Microcontroller is connected. A user selects a Pulse Count from the Pulse Count Drop Down Menu.
Results: Test Passed.

Test Case 3
Expected Results: When the Application is first launched or restarted, the Pulse Count Drop Down Menu setting shall be set to its default value as defined by the macro #define STARTUP_PULSE_COUNT 1
Inputs/State: The Application is launched or restarted.
Results: Test Passed.

"Power Level" Knob Testing
Test Case 1
Expected Results: When the Application is not connected to the Microcontroller and a user attempts to change the Power Level knob setting, the Application shall not send the new Power Level to the Microcontroller serial object. The Application shall reset the Power Level knob setting to the previous value and present the user with a dialogue box prompting the user to connect to the Microcontroller and ty again.
Inputs/State: The Application is not connected to the Microcontroller. A user changes the Power Level knob setting.
Results: Test Passed.

Test Case 2
Expected Results: When the Application is connected to the Microcontroller and a user changes the Power Level knob setting, the Application shall send the new Power Level value to the

Microcontroller serial object and present a dialogue box to the user confirming the new Power Level setting.
Inputs/State: The Application is connected to the Microcontroller. A user changes the Power Level knob setting.
Results: Test Passed.

Test Case 3
Expected Results: When the Application is first launched or restarted, the Power Level knob setting shall be set to its default value as defined by the macro #define STARTUP_POWER_LEVEL 1
Inputs/State: The Application is launched or restarted.
Results: Test Passed.

"Charge Capacitors" Button Testing
Test Case 1
Expected Results: When the Application is not connected to the Microcontroller and a user clicks the "Charge Capacitors" button, the Application shall present the user with a dialogue box prompting to connect to the Microcontroller and try again.
Inputs/State: The Application is not connected to the Microcontroller. The user clicks the "Charge Capacitors" button.
Results: Test Passed

Test Case 2
Expected Results: When the Application is connected to the Microcontroller and the user clicks the "Charge Capacitors" button, the Application shall send the charge capacitors command to the Microcontroller.
Inputs/State: The Application is connected to the Microcontroller. The user clicks the "Charge Capacitors" button.
Results: Test Passed.

Test Case 3
Expected Results: When the Application is connected to the Microcontroller and the user clicks the "Charge Capacitors" button, the "Pulse" button shall be disabled (grey, inactive, unclickable) on the GUI until the charge capacitors command returns successfully.
Inputs/State: The Application is connected to the Microcontroller. The user clicks the "Charge Capacitors" button.
Results: Test Passed.

Test Case 4

Expected Results: When the Application is connected to the Microcontroller and the user clicks the "Charge Capacitors" button, the "Pulse" button shall be disabled (grey, inactive, unclickable) on the GUI until the charge capacitors command returns successfully. If the charge capacitors command does not return successfully (returns and error) the "Pulse" button shall remain disabled and the Application shall present the user with a dialogue box with the message "Capacitor charging failed or timed out. Please manually reset the device."
Inputs/State: The Application is connected to the Microcontroller. The user clicks the "Charge Capacitors" button. The charge capacitors command does not return successfully.
Results: Test Passed.

4.32 Microcontroller Signal Testing

The requirements state that the TMS device should be capable of supplying up to a maximum of 10 pulses per minute to the coil at a maximum frequency of 36 Hq. The pulses delivered to the coil are directly controlled and shaped by the output from the microcontroller/op-amp(MC/OP) gate driver module. The pusles delivered to the coil should be 400 microseconds wide with a rise time t_r = 100 microseconds and fall time t_f = 100 microseconds. The rise and fall times were achieved by tuning the values of a passive resistor capacitor (RC) filter. The square wave output from the microcontroller is sent through the RC filter, for shaping, then through the op-amp to boost the voltage to the final level needed to drive the IGBT.

The frequency and number of pulses may be set by the user via the GUI. The frequency and number of pulses output by the MC/OP gate river module was verified using the 2024X Oscilloscope found in 5.4 Appendix III. Examples of the frequencies selected by the GUI and the output from the MC/OP gate driver are shown by the scope in figures 5, 6, and 7. Note that the cross hairs obscures one of the pulse waves because of the short time it occurs in.

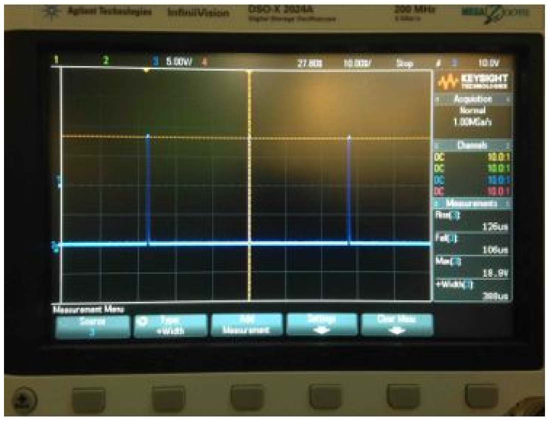


Fig. 5, 3 Pulses

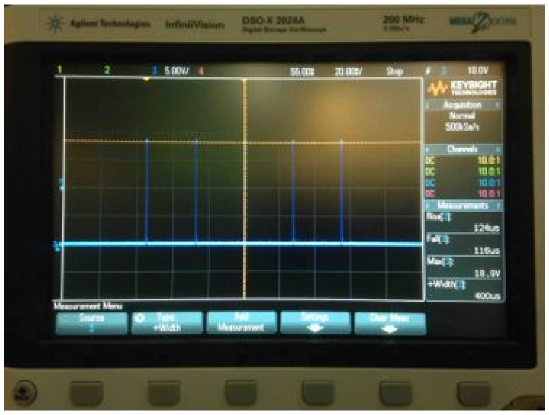
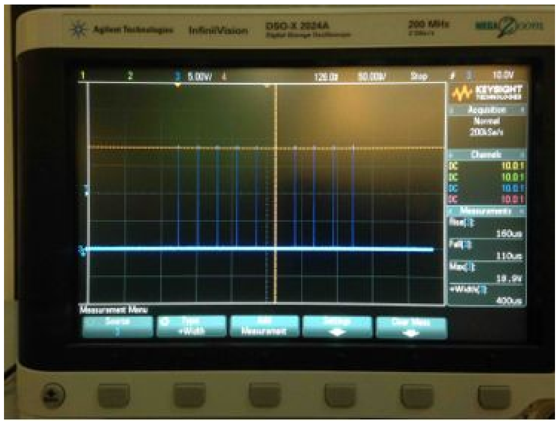


Fig. 6, 5 Pulses



These scope images show the number of pulses match the number selected by the GUI. The frequency measurement is not shown in these images, but the test was repeated with the frequency measurements on the scope and all frequencies matched the frequency selected by the GUI, between 1 and 36 Hz.

Fig. 7, 10 Pulses

The correct rise and fall times of the pulses were also verified with the DSO 2024X found in 5.4 Appendix III. The results are shown below in figure 8.

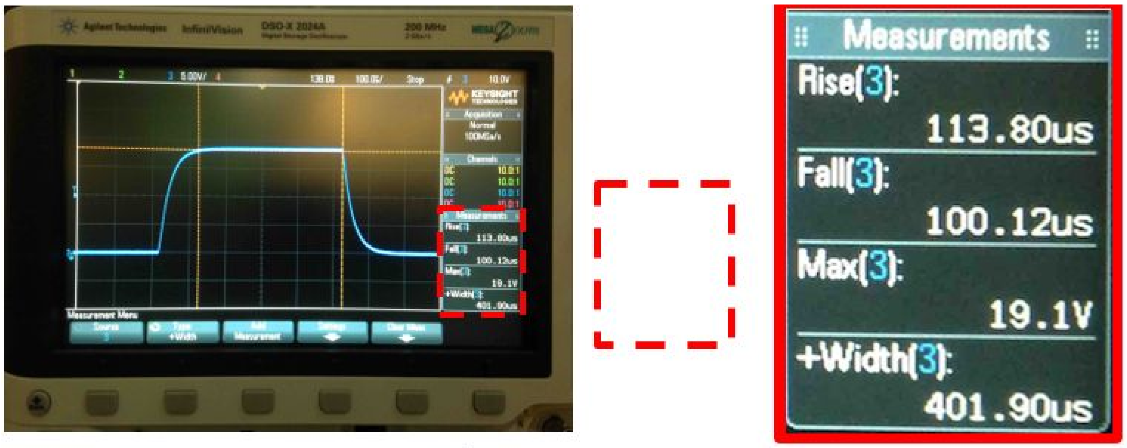


Fig. 8 Waveform of Pulse Sent to IGBT

4.33 Microcontroller/Hardware Interface Testing

In addition to controlling the IGBT gate to drive pulses to the coil, the microcontroller also interfaces with a capacitor charge level detection circuit and a switching relay to allow charging the capacitors and sensing charge level via the GUI.

The microcontroller controls charging of the capacitors by switching a relay placed between the output of the AC/DC transformer connected to wall power, and the capacitors. The DC output of the transformer is switched via the FeatherWing Adafruit Power Relay shown in figure 9. This relay is rated for 250V AC or DC and 5 A.

The microcontroller switches on the relay to charge the capacitors when the charge level sensed on the capacitors is below the charge level currently selected by the GUI and the user has issued a “charge capacitors” command via the GUI. Figure 9 below shows the charge detection and control components at a high level.

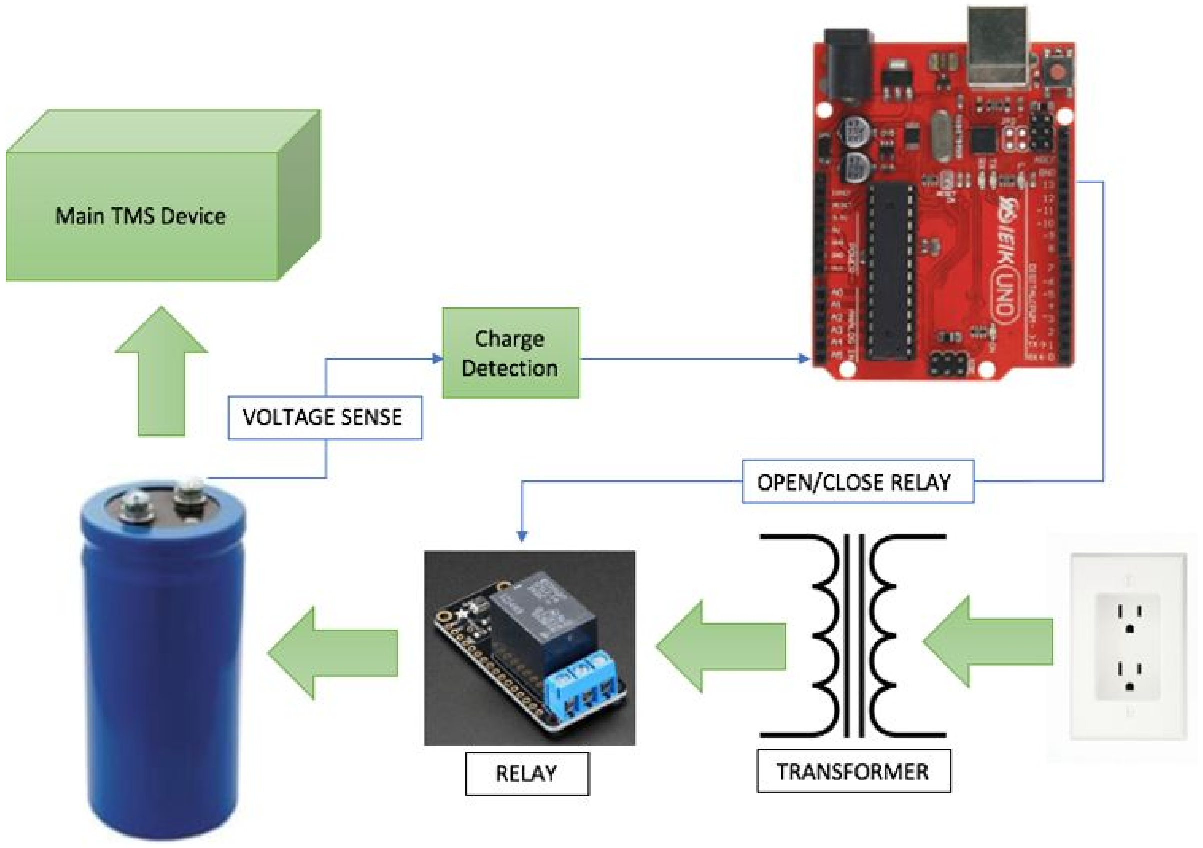


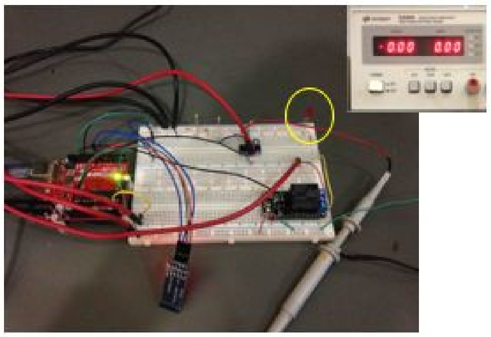
Fig. 9 High Level Component Diagram of Charge Control System

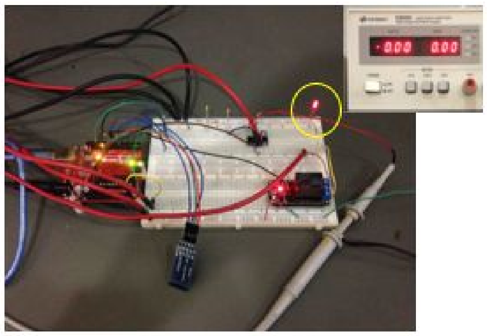
The capacitor charge detection circuit, charging control relay and microcontroller were tested as a unit before integration with the main TMS device. The testing was done using the Agilent E3631A DC Power Supply and the Keysight E3630A DC Power Supply shown in 5.4 Appendix III. The E3631 provided the supply voltage for the relay, while the E3630A provided a variable voltage to an analog input pin, the CAP_SENSE_PIN, on the microcontroller. The CAP_SENSE_PIN is used to monitor the capacitor charge level as indicated by the output of the charge detection circuit.

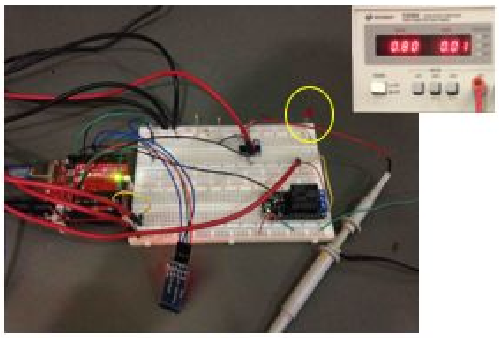
The relay was connected to switch 25V. This 25V simulates the DC output form the AC/DC transformer that would be switched to control charging the capacitors. A red light emitting diode (LED) was connected to the relay to visually indicate that the relay opens and closes according to the voltage on the CAP_SENSE_PIN. With the relay, capacitor charge detection circuit, and microcontroller connected, the E3630A output voltage to the CAP_SENSE_PIN was varied to simulate the voltage output by the capacitor charge detection circuit.

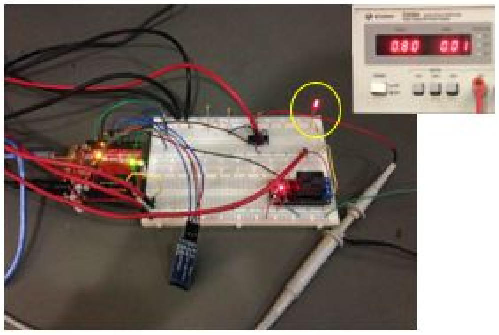
The relay switching and charge detection was tested for each of the three power level thresholds allowed by the GUI. when the “Charge Capacitors” command from the GUI was received by the microcontroller the relay was correctly opened. The benchtop voltage supply connected to the CAP_SENSE_PIN was then slowly increased. When this voltage increased to each of the three defined voltage thresholds set for the device’s three power levels, the relay closed correctly at each defined threshold, as shown below in the series of images in figure 10.

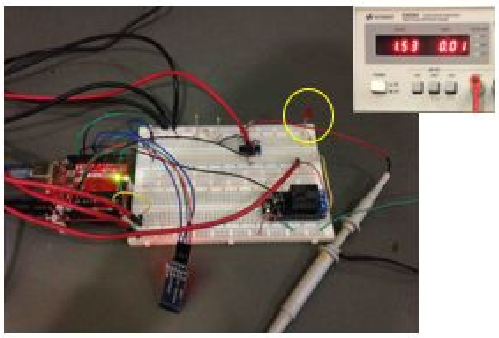
The images on the next page show that the relay opens (LED ON) when the “Charge Capacitors” command is given via the GUI and closes when the voltage on the CAP_SENSE_PIN reaches the threshold defined in the microcontroller code for the current charge level selected by the GUI.


1) 0 V on CAP_SENSE_PIN, Relay Closed, LED OFF


2) Charge to Level 1 command given: 0 V on CAP_SENSE_PIN, RELAY OPEN, LED ON


3) 0.80 V on CAP_SENSE_PIN (level 1 threshold) Relay Closed, LED OFF


4) Charge to Level 2 command given: 0.80 V on CAP_SENSE_PIN, RELAY OPEN, LED ON


5) 1.50 V on CAP_SENSE_PIN (level 2 threshold) Relay Closed, LED OFF

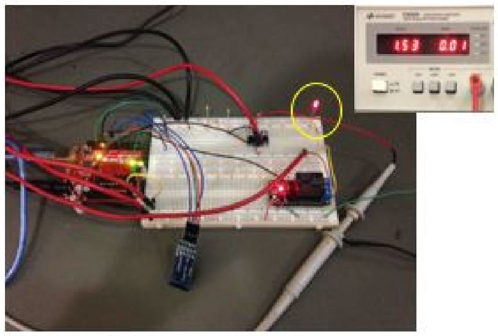

6) Charge to Level 3 command given: 1.50 V on CAP_SENSE_PIN, RELAY OPEN, LED ON

Fig. 10 Testing of Capacitor Sensing System

4.4 Functional Testing

Unit testing:
Our goals in the unit testing were the following;
- Microcontroller works correctly
- The rectifier circuit delivers the 240 Volts signal needed
- The power circuit has the correct behavior
- The operational amplifier functions within expectation in accordance to power supply
- Capacity charge circuit works as expected

Each individual components are working within expectations. Each component is within the limits of our calculation.

Integration testing:
Our goals in the integration testing were the following;

- The microcontroller monitors the correct parts of the circuit
- The microcontroller controls the IGBT’s in the right manner.
- The rectifier circuit takes the wall input signal and converts it to the desired output
- The capacitors in the power circuit charge correctly with the signal sent from the rectifier circuit
- The diodes in the power circuit activate and deactivate when anticipated with the signal from the rectifier circuit
- Current flows through device to coils and outputs current

Each component is working as expected with each other after extensive calculation. In the beginning, many issues occured that we had to rectify. Currently everything is now working together.

Acceptance testing:
Our goals in the acceptance testing were the following;

- Test that the current pulse being delivered is the current expected
- Check that the time the pulse is delivered is about 400 µs
- Test the rise and fall times of the pulse are within 100 µs
- Each IGBT outputs ~320-340 amps

As of now, we are finally obtaining the outputs we expected and we can increase the current by changing the specs of our coil.

4.5 Non-Functional Testing

When building our project, we took the following non-functional considerations on mind:

1- User friendly GUI.
After designing our GUI, we invited a couple of students who are not majoring in science or engineering to make sure that the interface is user friendly.

2- heavy duty chassis.
Testing this was easy. We knew the weights of all of our components, we simply put a similar weight in our box before building our project to make sure that the chassis can handle it.

3- Built in Voltmeter
We included a voltmeter on the wall of our box. To make sure that the voltmeter is accurate even at a high temperature, we used an outside voltmeter and compared the results multiple times during different operation stages.

4- fans to increase circuit cooling

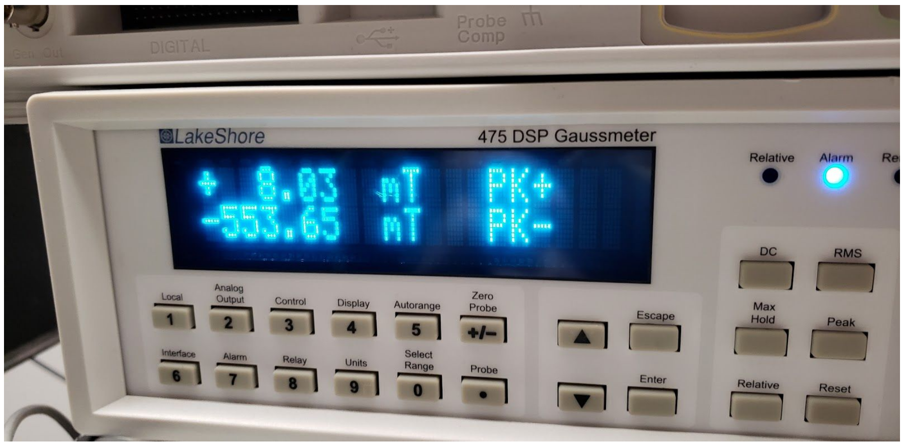
To test this, we simply operated our design multiple times, and ensured that the device does not get heated.

5- less expensive than similar generators in the market
We ran a market survey for the price of similar products in the market, and compared it with our total cost including labor cost. Since what is available in the market is to be used on human, we compared the price per a unit of magnetic field instead of comparing the products as a whole.

4.6 Results

After almost an entire semester of trial and error testing, trying to determine the best and most efficient ways to test our circuit, we received great results. First we sent through a smaller signal in the IGBT gate with low voltage from the capacitors and saw the behavior of the output through an oscilloscope. These results were good, we saw the expected behavior we were hoping to witness in our circuit. Then we received results that did not seem correct once we tested with individual resistors to find the current flowing through the inductor. Those resistors would limit the current and not let it reach its full potential. With the help of our advisors we came up with the idea of using a very long wire with very small resistance so the current wouldn't be limited as much. It still gave results much lower than what the circuit was design to perform.

The last testing method we used was a gauss meter which proved to be quite efficient. Once we set it up we recorded much higher current signals than we did with the resistors and the wire. With this set up proving to be effective we decided to add the gate voltage booster signal and charge the capacitors to the full and see if our main objective - getting about 2,000 Amperes of current - would be a success. Once the pulse went through we measured about .56 Teslas as seen in fig. [[[]]] and our gator coil connected to the current blew up. The amount of teslas we got calculated to be about 1,970 Amperes which leads us to call this project a success.



5 Closing Materials

5.1 Conclusion

This project has shown that designing and building a modular, cost-effective high current pulse generating device is feasible. Despite a small number of component failures and time constraints, we achieved the fundamental requirements of our design: current pulses of nearly 2000 Amps sustained for 400 microseconds.

5.2 Appendix I Operation Manual

The TMS GUI is designed to be robust and failsafe against entering unknown states. The GUI makes extensive use of the Matlab *waitfor()* and *msgbox()* methods to present the user with confirmation after any setting is changed. The GUI is also designed with interrupts canceled and GUI items temporarily disabled as needed. This ensures that if a user tries to click a GUI item while another action is in progress, the interrupt action is not queued for later execution, but canceled (i.e. ignored) instead. This makes the GUI behavior more deterministic and secure.

The steps for basic operation of the TMS device via the GUI are described below. **Note:** The following steps assume that the physical device has been set up correctly and the embedded microcontroller/Bluetooth unit is powered on and the ready to accept a connection from the GUI. The following also assumes that Matlab is installed on your computer. The TMS GUI was developed on *Matlab r2017b 64-bit*. Compatibility with other releases of Matlab is expected, but was not tested.

0. Download the Matlab TMS Application from the Team 4 Senior Design website. Go to http://sddec18-04.sd.ece.iastate.edu/docs.html and click on "Download Matlab TMS Application".

1. Launch Matlab and navigate to the folder where the TMS.mlapp application was downloaded.

2. Right click on the TMS.mlapp in the file explorer window and click "Run" as seen in figure 11.

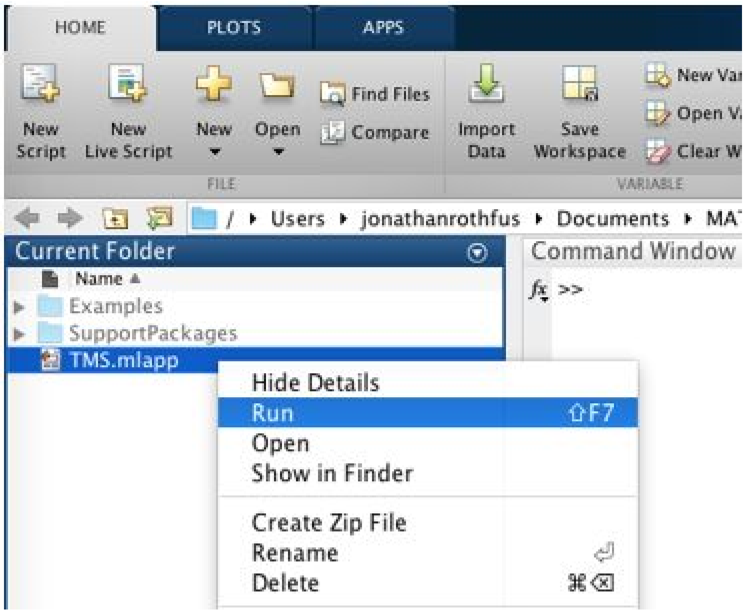
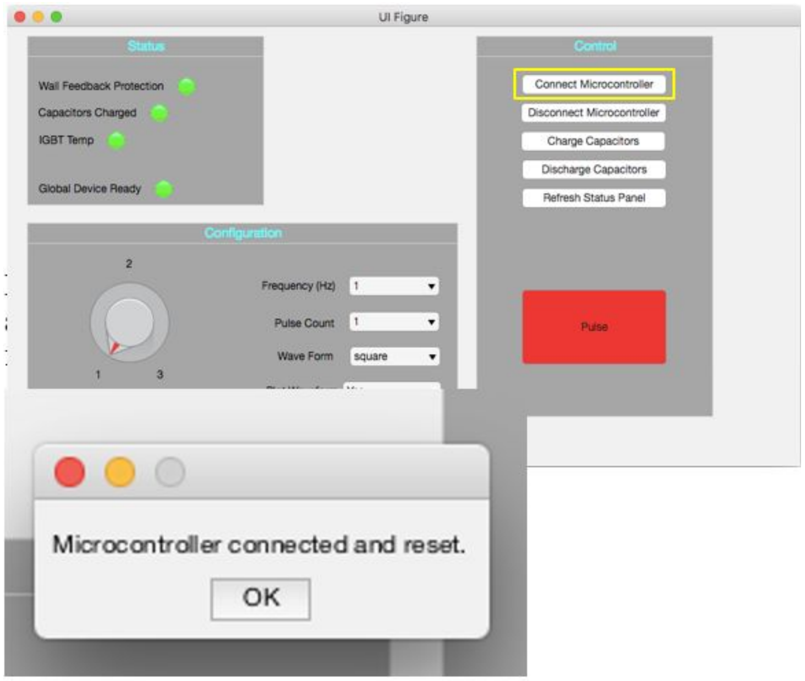


Fig. 11

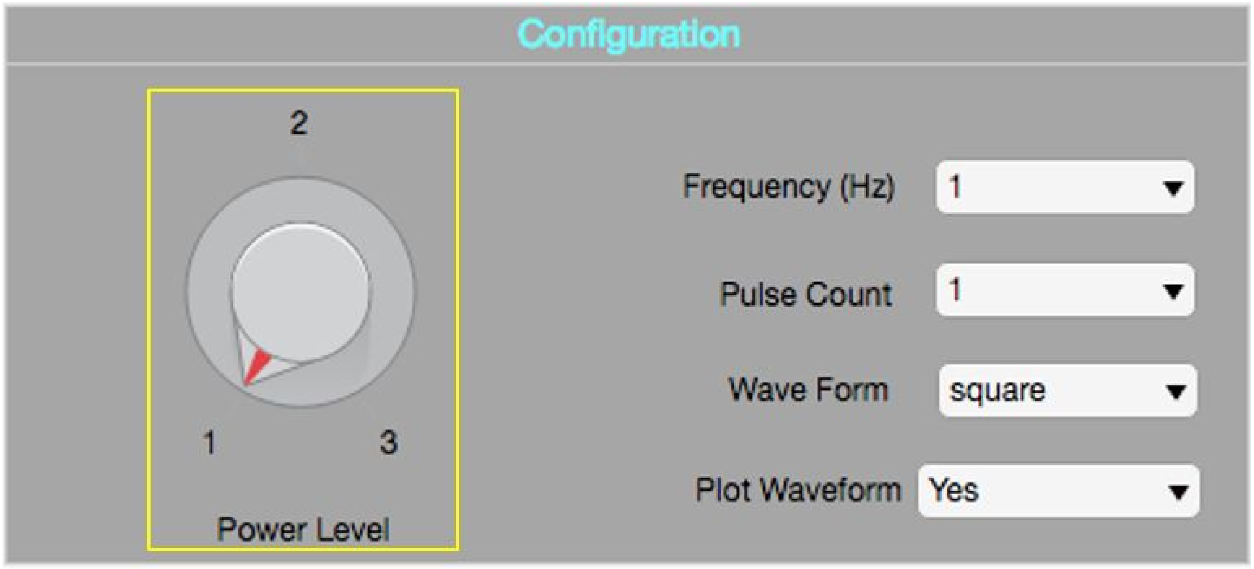
3. When the GUI launches, click the "Connect Microcontroller" button. In a few seconds, the GUI should present a dialogue box confirming microcontroller connection as shown below in figure 12. If connection fails, a dialogue box will also be presented stating connection failed.



Microcontroller connected and reset.

Fig. 12

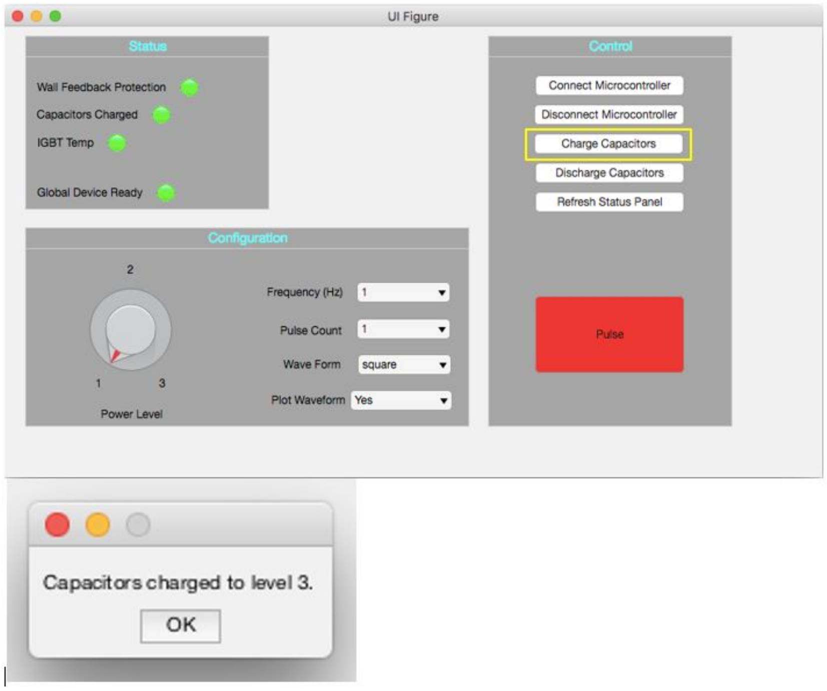
4. Select a power level with the Power Level knob setting as seen in figure 13. The startup default level is 1. The Power Level setting does not affect pulse power, but it determines how many pulses may be fired before recharging the capacitors is necessary. If you intend to fire a large number of pulses (>30) choose Power Level 3.



Frequency (Hz) 1
Pulse Count 1
Wave Form square
Plot Waveform Yes

Fig. 13

5. Click the "Charge Capacitors" button to begin charging the device as seen in figure 14.When the capacitors have charged to the Power Level selected, a dialogue box will pop up to confirm charging is completed. If charging fails or times out for any reason, a dialogue box will also be presented stating charging failed or timed out.



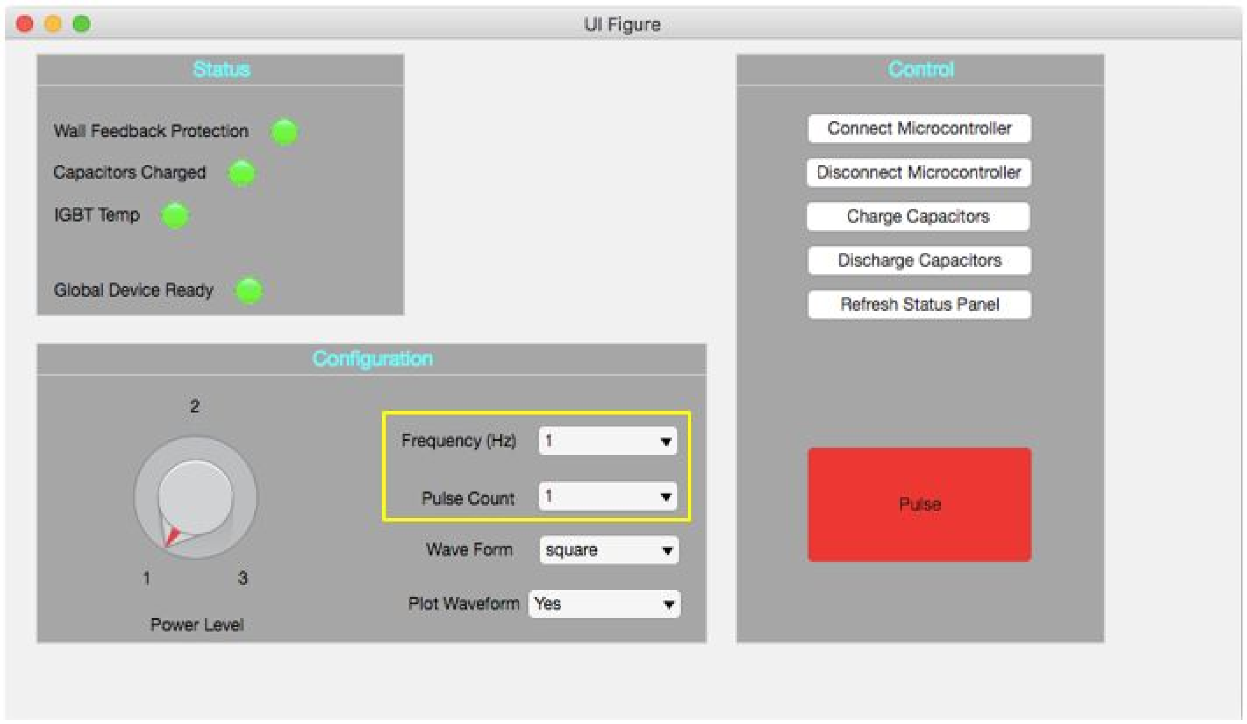
Frequency (Hz) 1
Pulse Count 1
Wave Form square
Plot Waveform Yes

Capacitors charged to level 3.

Fig. 14

6. Select a Frequency and Pulse Count from the drop down menus as seen in figure 15. The default value for both of the fields is 1. Note that the maximum pulse count allowed is 10. If a

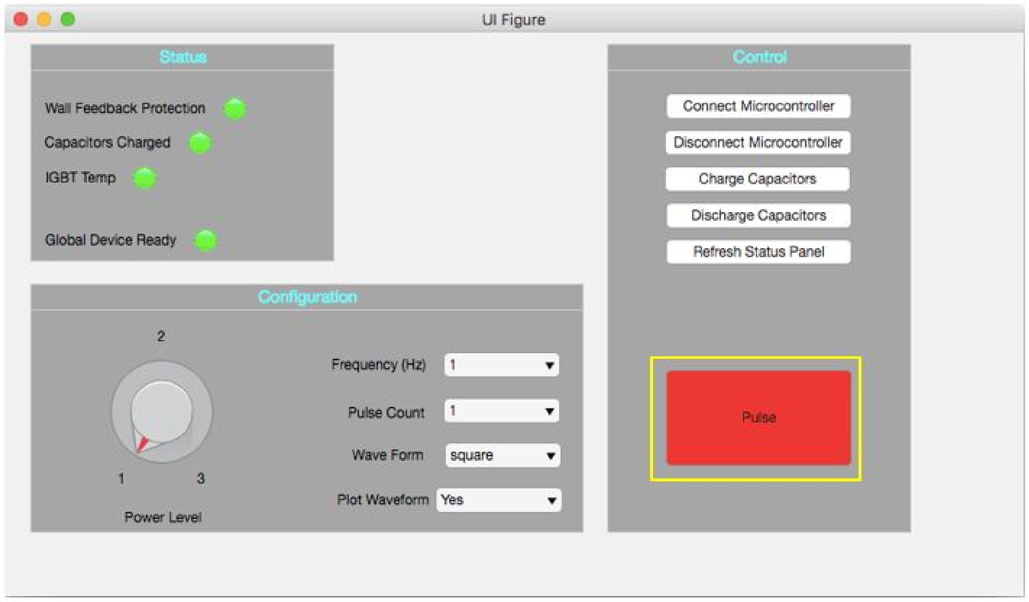
Frequency of greater than 10 Hz is chosen the device will fire pulses at the chosen frequency until 10 pulses have been fired, then stop.



Frequency (Hz) 1
Pulse Count 1
Wave Form square
Plot Waveform Yes

Fig. 15

7. The device is now ready to fire pulses. Ensure the physical device and coil are placed properly and the area is safe for pulsing. Click the red "Pulse" button as seen in figure 16.



Frequency (Hz) 1
Pulse Count 1
Wave Form square
Plot Waveform Yes

Fig. 16

8. Pulsing is complete. Each time the pulse button is clicked the capacitor charge level is checked before firing the pulse. If the capacitors need to be recharged the pulse command will not be sent to the device and a dialogue box will instead prompt the user to recharge the capacitors as described in steps **4-5** above.

5.3 Appendix II Past Design Analysis

The past thyristor design as seen in figure 17 was inspired by Polson's Patent, Magnetic Simulator for Neuro-Muscular Tissues. In the design there is an electrical storage component, capacitor, and a switching device, thyristor, that controls the flow of current to an inductive load [6]. One of the unique features of this design is the regenerative circuit. This portion of the circuit recaptures the electrical energy dumped into the load in capacitor storage, Cr, rather than having it return to the capacitors, Cs.

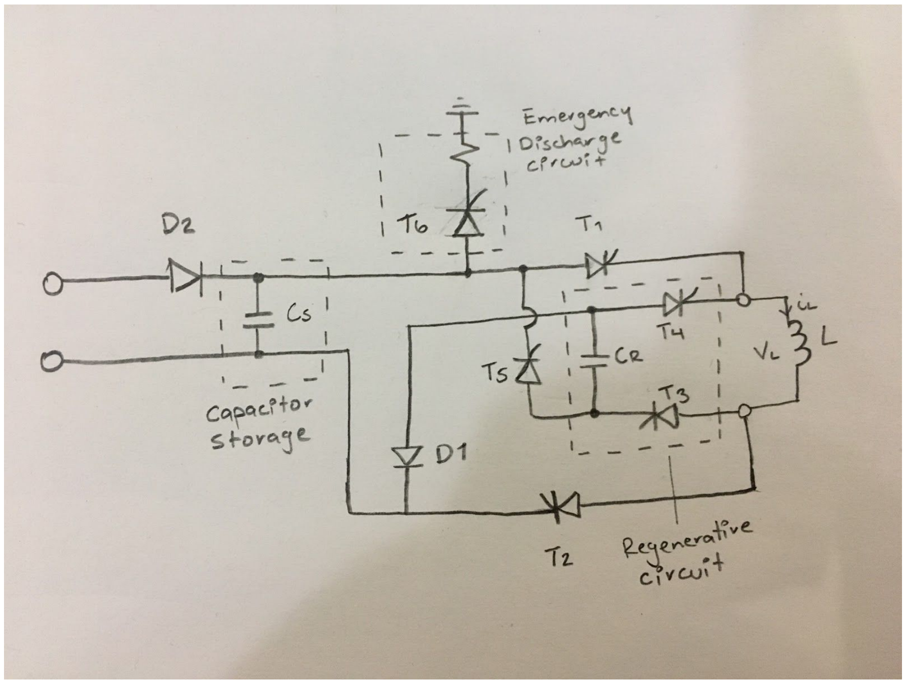


Fig. 17

This design was dependent of having a thyristor that could handle the back electromotive force (EMF). The team had chosen the thyristor IRK 230-20 for our design, and began requesting quotes from several high-power electrical component suppliers. The most promising of the companies the team was in contact with was 5s components, and still so many problems arose with securing the part the decision to go with a design not thyristor dependent was made.

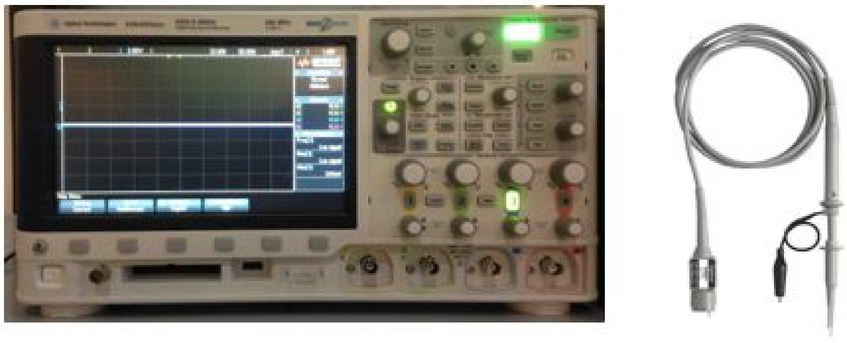
Other senior design teams in the past have developed and built plans for a TMS HCPG. Their machines were designed to reach a peak current of 1000 amperes, a pulse width between 50-400 microseconds, have an easy to use GUI, and a budget of $500 [4,5]. They also used an IGBT in their design, and a GUI based in Matlab. We were able to learn from their shortcomings on component failure. Their IGBTs were expensive and it was a large hit to the budget whenever one would blow. We sourced IGBTs that were only $150 and prepared to have failures in our budget.

5.4 Appendix III Other Considerations

Test Equipment Used

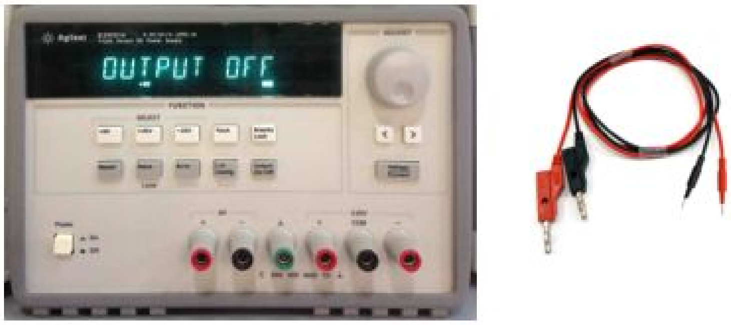
Basic functionality of the embedded microcontroller used for this project and all related hardware testing described and shown below in figure 18 was done using the following standard electronic test equipment found in the Iowa State University’s Transformative Learning Area.

Agilent DSO-X 2024A Digital Storage Oscilloscope with 10:1 Passive Probe



Keysight 34410A Multimeter with Banana-to-Breadboard Connectors

Keysight E3631A DC Power Supply with Banana-to-Breadboard Connectors



Keysight E3630A DC Power Supply with Banana-to-Breadboard Connectors

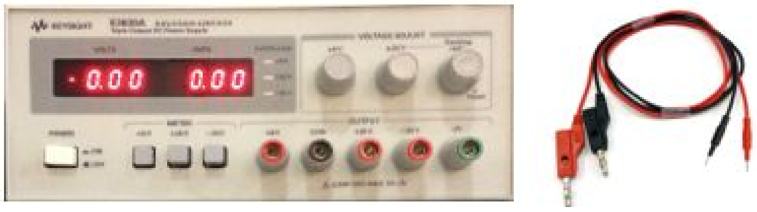


Fig. 18 Testing Equipment

5.5 Appendix IV Code

TMS.mlapp Matlab Code

This code can also be viewed in Matlab by downloading the TMS.mlapp from the URL given above in "GUI Operation Steps" step **0.** above.

It's much prettier viewed in Matlab.

```
classdef TMS < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
UIFigure                         matlab.ui.Figure
    StatusPanel                  matlab.ui.container.Panel
    CapacitorsChargedLampLabel   matlab.ui.control.Label
    CapacitorsChargedLamp        matlab.ui.control.Lamp
    WallFeedbackProtectionLampLabel  matlab.ui.control.Label
    WallFeedbackProtectionLamp   matlab.ui.control.Lamp
    IGBTTempLampLabel            matlab.ui.control.Label
    IGBTTempLamp                 matlab.ui.control.Lamp
    GlobalDeviceReadyLampLabel   matlab.ui.control.Label
    GlobalDeviceReadyLamp        matlab.ui.control.Lamp
    ConfigurationPanel           matlab.ui.container.Panel
    WaveFormDropDownLabel        matlab.ui.control.Label
    WaveFormDropDown             matlab.ui.control.DropDown
    PowerLevelKnobLabel          matlab.ui.control.Label
    PowerLevelKnob               matlab.ui.control.Knob
    PlotWaveformDropDownLabel    matlab.ui.control.Label
    PlotWaveformDropDown         matlab.ui.control.DropDown
    PulseCountDropDownLabel      matlab.ui.control.Label
    PulseCountDropDown           matlab.ui.control.DropDown
    FrequencyHzDropDownLabel     matlab.ui.control.Label
    FrequencyHzDropDown          matlab.ui.control.DropDown
    ControlPanel                 matlab.ui.container.Panel
    connect_mc                   matlab.ui.control.Button
    charge_caps                  matlab.ui.control.Button
    pulse                        matlab.ui.control.Button
    discharge_caps               matlab.ui.control.Button
    disconnect_mc                matlab.ui.control.Button
    refresh_status               matlab.ui.control.Button
    end

    %                                             %
    % Private app properties                      %
    %                                             %
    properties (Access = private)
                                 Serial; % Serial port object
    MAX_PULSE_COUNT % Max number of pulses to allow per pulse command, per requirements
    MIN_PULSE_COUNT % Minimum number of pulses to allow per pulse command, a derived requirement
    MAX_FREQUENCY % Max frequency allowed, per requirements
    MIN_FREQUENCY % Minimum frequency to allow, a derived requirement
    STARTUP_PULSE_COUNT % Value of pulse count when app starts or resets
    STARTUP_FREQUENCY % Value of frequecny when app starts or resets
    STARTUP_POWER_LEVEL % Power level setting when app starts or resets
    TIMEOUT % Seconds to wait for response from Microcontroller before reporting an error
    E_CAP_CHARGE_TIMEOUT % Error to indicate capacitor charging failed
    end

    %                                          %
    % Private utility methods   %
    %                                          %
    methods (Access = private)

    % Blink the UI to bring main app window back into focus after a msgbox
                              function blink_UI(app)
                         app.UIFigure.Visible = 'off';
                          app.UIFigure.Visible = 'on';
                                               return;
                                                  end

                           % Initialize serial connection
                             function init_serial(app)
                                  delete(instrfindall);
          app.Serial = serial('/dev/tty.lilblue-DevB');
                            app.Serial.BaudRate = 9600;
      app.Serial.Terminator = 'LF'; %(newline, ascii 10)
                                               return;
                                                  end

                             % Initialize parameter limits
                               function init_limits(app)
                     app.MAX_PULSE_COUNT = 10;
                        app.MIN_PULSE_COUNT = 1;
                       app.MAX_FREQUENCY = 36;
                          app.MIN_FREQUENCY = 1;
                                               return;
        end

                   % Initialize parameter default values
                             function init_defaults(app)
              app.STARTUP_PULSE_COUNT = 1;
                 app.STARTUP_FREQUENCY = 1;
             app.STARTUP_POWER_LEVEL = 1;
                               app.TIMEOUT = 60;
        app.E_CAP_CHARGE_TIMEOUT = 101;
                                               return;
                                                  end
                              % Reset app to known state
                                     function reset(app)
    app.PulseCountDropDown.Value = num2str(app.STARTUP_PULSE_COUNT);
    app.FrequencyHzDropDown.Value = num2str(app.STARTUP_FREQUENCY);
    app.PowerLevelKnob.Value = double(app.STARTUP_POWER_LEVEL);
                                                 return;
                                                    end

                   function ready = device_ready(app)
                   if strcmp(app.Serial.Status, 'closed')
                                               ready = 0;
                                                    end
                                 fprintf(app.Serial, 's');
          power_level = uint8(fread(app.Serial,1));
            frequency = uint8(fread(app.Serial,1));
          pulse_count = uint8(fread(app.Serial,1));
                                               ready = 1;
    end

  end


  methods (Access = private)

     % Code that executes after component creation
                                 function startupFcn(app)
            % Define rgb states of indicator lamps.
%                                        global GREEN;
%                                     GREEN = [0, 1, 0];
%                                        global WHITE;
%                                     WHITE = [1, 1, 1];
              app.UIFigure.Interruptible = 'off';
              app.UIFigure.BusyAction  = 'cancel';
                                      init_serial(app);
                                      init_limits(app);
                                    init_defaults(app);
      reset(app);


%       % Alias indicator lamp names to shorten and initialize all to OFF.
%                                    global cap_lamp;
%         cap_lamp = app.CapacitorsChargedLamp;
%                             cap_lamp.Color = WHITE;
%                                   global igbt_lamp;
%            igbt_lamp = app.IGBTTempLamp;
%            igbt_lamp.Color = WHITE;
%            global wall_lamp;
%    wall_lamp = app.WallFeedbackProtectionLamp;
%            wall_lamp.Color = WHITE;
%            global global_lamp;
%    global_lamp = app.GlobalDeviceReadyLamp;
%            global_lamp.Color = WHITE;
                                        end

        % Button pushed function: connect_mc
        function connect_mc_button(app, event)
            if strcmp(app.Serial.Status, 'closed')
                fopen(app.Serial);
            if strcmp(app.Serial.Status, 'closed')
        waitfor(msgbox('Microcontroller connection failed.'));
                blink_UI(app);
            elseif strcmp(app.Serial.Status, 'open')
                fprintf(app.Serial, 'r');
            response = fread(app.Serial,1);
                if response == 1
        waitfor(msgbox('Microcontroller connected and reset.'));
                blink_UI(app);
                else
        waitfor(msgbox('Microcontroller connected but reset FAILED.'));
                blink_UI(app);
                end
                end
                else
    waitfor(msgbox('Microcontroller connected.'));
                blink_UI(app);
                end
                end

        % Button pushed function: pulse
        function pulseButtonPushed(app, event)
            if strcmp(app.Serial.Status, 'open')
                fprintf(app.Serial, 'p');
                return;
                else
    waitfor(msgbox('Please connect to microcontroller and try again.'));
                blink_UI(app);
                return;
                end
                end
```

```
% Button pushed function: disconnect_mc
function disconnect_mcButtonPushed(app, event)
fclose(app.Serial);
if strcmp(app.Serial.Status, 'closed')
waitfor(msgbox('Microcontroller disconnected.'));
blink_UI(app);
else
waitfor(msgbox('Microcontroller could not be disconnected.'));
blink_UI(app);
end
end

% Button pushed function: refresh_status
function refresh_statusButtonPushed(app, event)
waitfor(msgbox('This selection is not enabled.'));
blink_UI(app);
return;
end

% Button pushed function: charge_caps
function charge_capsButtonPushed(app, event)
if strcmp(app.Serial.Status, 'closed')
waitfor(msgbox('Please connect to microcontroller and try again.'));
blink_UI(app);
return;
end
flushinput(app.Serial);
%disp(app.Serial.BytesAvailable); %debug
app.pulse.Enable = 'off';
fprintf(app.Serial, 'c');
tic;
while (1)
if (app.Serial.BytesAvailable > 0)
break;
elseif (toc >= app.TIMEOUT)
break;
end
end

if (app.Serial.BytesAvailable > 0)
response_val = int8(fread(app.Serial,1));
if response_val == int8(round(app.PowerLevelKnob.Value))
waitfor(msgbox(sprintf('Capacitors charged to level %d.', response_val)));
app.pulse.Enable = 'on';
```

```
                                        blink_UI(app);
        elseif response_val == app.E_CAP_CHARGE_TIMEOUT
            waitfor(msgbox('Capacitor charging failed. Please manually reset the device'));
                                        blink_UI(app);
                                                end
                                                else
        waitfor(msgbox('No response from Microcontroller. Please manually reset the device'));
                                        blink_UI(app);
                                                end
                                            return;
                                                end

    % Value changed function: PowerLevelKnob
function PowerLevelKnobValueChanged(app, event)
                    if strcmp(app.Serial.Status, 'closed')
        waitfor(msgbox('Please connect to microcontroller and try again.'));
        app.PowerLevelKnob.Value = double(app.STARTUP_POWER_LEVEL);
                                        blink_UI(app);
                                            return;
                                                end
    value = string(round(app.PowerLevelKnob.Value));
                        command_str = strcat('l',value);
                        fprintf(app.Serial, command_str);
        response_val = double(fread(app.Serial,1));
                    if response_val == str2double(value)
        waitfor(msgbox(sprintf('Power level set to %s.', value)));
                                                else
        waitfor(msgbox('Power level not successfully set.'));
                                                end
                                        blink_UI(app);
                                            return;
                                                end

% Value changed function: PulseCountDropDown
function PulseCountDropDownValueChanged(app, event)
                    if strcmp(app.Serial.Status, 'closed')
        waitfor(msgbox('Please connect to microcontroller and try again.'));
        app.PulseCountDropDown.Value = num2str(app.STARTUP_PULSE_COUNT);
                                        blink_UI(app);
                                            return;
                                                end
    value = string(app.PulseCountDropDown.Value);
                        command_str = strcat('n', value);
                        fprintf(app.Serial, command_str);
        response_val = double(fread(app.Serial,1));
    %wait here for response from MC: either OK or ERROR
                if response_val == str2double(value)
        waitfor(msgbox(sprintf('Pulse count set to %s.', value)));
                                                blink_UI(app);
                                                                else
        waitfor(msgbox('Pulse count not successfully set.'));
                                                blink_UI(app);
                                                                end
                                                            return;
                                                                end

% Value changed function: FrequencyHzDropDown
function FrequencyHzDropDownValueChanged(app, event)
                if strcmp(app.Serial.Status, 'closed')
        app.FrequencyHzDropDown.Value = num2str(app.STARTUP_FREQUENCY);
        waitfor(msgbox('Please connect to microcontroller and try again.'));
                                                blink_UI(app);
                                                            return;
                                                                end
    value = string(app.FrequencyHzDropDown.Value);
                        command_str = strcat('f', value);
                        fprintf(app.Serial, command_str);
        response_val = double(fread(app.Serial,1));

                if response_val == str2double(value)
        waitfor(msgbox(sprintf('Frequency set to %s.', value)));
                                                blink_UI(app);
                                                                else
        waitfor(msgbox('The value was not sucessfully set.'));
                                                blink_UI(app);
                                                                end
                                                            return;
                                                                end

        % Button pushed function: discharge_caps
function discharge_capsButtonPushed(app, event)
                if strcmp(app.Serial.Status, 'closed')
        waitfor(msgbox('Please connect to microcontroller and try again.'));
                                                blink_UI(app);
                                                            return;
                                                                end
    waitfor(msgbox('This selection is not enabled.'));
                                                blink_UI(app);
                                                            return;
                                                                end
        % Value changed function: PlotWaveformDropDown
    function PlotWaveformDropDownValueChanged(app, event)
        waitfor(msgbox('This selection is not enabled.'));
                                                blink_UI(app);
                                                        return;
                                                            end

    % Value changed function: WaveFormDropDown
    function WaveFormDropDownValueChanged(app, event)
        waitfor(msgbox('This selection is not enabled.'));
                                                blink_UI(app);
                                                        return;
                                                            end
    end

    % App initialization and construction
    methods (Access = private)

                    % Create UIFigure and components
                        function createComponents(app)

                                            % Create UIFigure
                                    app.UIFigure = uifigure;
            app.UIFigure.Position = [100 100 871 480];
                        app.UIFigure.Name = 'UI Figure';

          % Create StatusPanel
            app.StatusPanel = uipanel(app.UIFigure);
          app.StatusPanel.ForegroundColor = [0 1 1];
           app.StatusPanel.TitlePosition = 'centertop';
                      app.StatusPanel.Title = 'Status';
        app.StatusPanel.BackgroundColor = [0.651 0.651 0.651];
                      app.StatusPanel.FontSize = 14;
         app.StatusPanel.Position = [23 286 260 185];

              % Create CapacitorsChargedLampLabel
        app.CapacitorsChargedLampLabel = uilabel(app.StatusPanel);
            app.CapacitorsChargedLampLabel.HorizontalAlignment = 'right';
            app.CapacitorsChargedLampLabel.Position = [7 94 113 15];
        app.CapacitorsChargedLampLabel.Text = 'Capacitors Charged';

                     % Create CapacitorsChargedLamp
        app.CapacitorsChargedLamp = uilamp(app.StatusPanel);
        app.CapacitorsChargedLamp.Position = [135 91 20 20];
```

```
% Create WallFeedbackProtectionLampLabel
app.WallFeedbackProtectionLampLabel = uilabel(app.StatusPanel);
    app.WallFeedbackProtectionLampLabel.HorizontalAlignment = 'right';
    app.WallFeedbackProtectionLampLabel.Position = [7 123 143 15];
app.WallFeedbackProtectionLampLabel.Text = 'Wall Feedback Protection';

        % Create WallFeedbackProtectionLamp
app.WallFeedbackProtectionLamp = uilamp(app.StatusPanel);
    app.WallFeedbackProtectionLamp.Position = [165 120 20 20];

                    % Create IGBTTempLampLabel
app.IGBTTempLampLabel = uilabel(app.StatusPanel);
    app.IGBTTempLampLabel.HorizontalAlignment = 'right';
app.IGBTTempLampLabel.Position = [7 64 66 15];
app.IGBTTempLampLabel.Text = 'IGBT Temp';

    % Create IGBTTempLamp
app.IGBTTempLamp = uilamp(app.StatusPanel);
app.IGBTTempLamp.Position = [88 61 20 20];

      % Create GlobalDeviceReadyLampLabel
app.GlobalDeviceReadyLampLabel = uilabel(app.StatusPanel);
    app.GlobalDeviceReadyLampLabel.HorizontalAlignment = 'right';
    app.GlobalDeviceReadyLampLabel.Position = [7 11 118 15];
app.GlobalDeviceReadyLampLabel.Text = 'Global Device Ready';

               % Create GlobalDeviceReadyLamp
app.GlobalDeviceReadyLamp = uilamp(app.StatusPanel);
app.GlobalDeviceReadyLamp.Position = [140 8 20 20];

                         % Create ConfigurationPanel
app.ConfigurationPanel = uipanel(app.UIFigure);
app.ConfigurationPanel.ForegroundColor = [0 1 1];
    app.ConfigurationPanel.TitlePosition = 'centertop';
app.ConfigurationPanel.Title = 'Configuration';
app.ConfigurationPanel.BackgroundColor = [0.651 0.651 0.651];
           app.ConfigurationPanel.FontSize = 14;
app.ConfigurationPanel.Position = [23 55 473 212];

            % Create WaveFormDropDownLabel
app.WaveFormDropDownLabel = uilabel(app.ConfigurationPanel);
app.WaveFormDropDownLabel.HorizontalAlignment = 'right';
app.WaveFormDropDownLabel.Position = [270 59 66 15];
app.WaveFormDropDownLabel.Text = 'Wave Form';
```

```
                    % Create WaveFormDropDown
        app.WaveFormDropDown = uidropdown(app.ConfigurationPanel);
        app.WaveFormDropDown.Items = {'square', 'triangle', 'sinusoid'};
            app.WaveFormDropDown.ValueChangedFcn = createCallbackFcn(app,
@WaveFormDropDownValueChanged, true);
        app.WaveFormDropDown.Position = [354 55 100 22];
            app.WaveFormDropDown.Value = 'square';

                    % Create PowerLevelKnobLabel
        app.PowerLevelKnobLabel = uilabel(app.ConfigurationPanel);
        app.PowerLevelKnobLabel.HorizontalAlignment = 'center';
        app.PowerLevelKnobLabel.Position = [78 6 71 15];
        app.PowerLevelKnobLabel.Text = 'Power Level';

                        % Create PowerLevelKnob
        app.PowerLevelKnob = uiknob(app.ConfigurationPanel, 'continuous');
                    app.PowerLevelKnob.Limits = [1 3];
            app.PowerLevelKnob.MajorTicks = [1 2 3];
        app.PowerLevelKnob.ValueChangedFcn = createCallbackFcn(app, @PowerLevelKnobValueChanged,
true);
                app.PowerLevelKnob.MinorTicks = [];
        app.PowerLevelKnob.Position = [68 59 88 88];
                    app.PowerLevelKnob.Value = 1;

                % Create PlotWaveformDropDownLabel
        app.PlotWaveformDropDownLabel = uilabel(app.ConfigurationPanel);
            app.PlotWaveformDropDownLabel.HorizontalAlignment = 'right';
            app.PlotWaveformDropDownLabel.Position = [257 21 84 15];
        app.PlotWaveformDropDownLabel.Text = 'Plot Waveform';

                    % Create PlotWaveformDropDown
        app.PlotWaveformDropDown = uidropdown(app.ConfigurationPanel);
            app.PlotWaveformDropDown.Items = {'Yes', 'No'};
            app.PlotWaveformDropDown.ValueChangedFcn = createCallbackFcn(app,
@PlotWaveformDropDownValueChanged, true);
        app.PlotWaveformDropDown.Position = [346 17 109 22];
            app.PlotWaveformDropDown.Value = 'No';

                % Create PulseCountDropDownLabel
        app.PulseCountDropDownLabel = uilabel(app.ConfigurationPanel);
            app.PulseCountDropDownLabel.HorizontalAlignment = 'right';
        app.PulseCountDropDownLabel.Position = [266 95 71 15];
        app.PulseCountDropDownLabel.Text = 'Pulse Count';
```

```
                    % Create PulseCountDropDown
        app.PulseCountDropDown = uidropdown(app.ConfigurationPanel);
        app.PulseCountDropDown.Items = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10'};
        app.PulseCountDropDown.ItemsData = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10'};
            app.PulseCountDropDown.ValueChangedFcn = createCallbackFcn(app,
@PulseCountDropDownValueChanged, true);
        app.PulseCountDropDown.Position = [353 93 100 22];
                app.PulseCountDropDown.Value = '1';

                % Create FrequencyHzDropDownLabel
        app.FrequencyHzDropDownLabel = uilabel(app.ConfigurationPanel);
        app.FrequencyHzDropDownLabel.HorizontalAlignment = 'right';
        app.FrequencyHzDropDownLabel.Position = [252 136 86 15];
        app.FrequencyHzDropDownLabel.Text = 'Frequency (Hz)';

                    % Create FrequencyHzDropDown
        app.FrequencyHzDropDown = uidropdown(app.ConfigurationPanel);
        app.FrequencyHzDropDown.Items = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17',
'18', '19', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36'};
        app.FrequencyHzDropDown.ItemsData = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16',
'17', '18', '19', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36'};
            app.FrequencyHzDropDown.ValueChangedFcn = createCallbackFcn(app,
@FrequencyHzDropDownValueChanged, true);
        app.FrequencyHzDropDown.Position = [353 132 100 22];
               app.FrequencyHzDropDown.Value = '1';

                            % Create ControlPanel
            app.ControlPanel = uipanel(app.UIFigure);
         app.ControlPanel.ForegroundColor = [0 1 1];
          app.ControlPanel.TitlePosition = 'centertop';
                      app.ControlPanel.Title = 'Control';
        app.ControlPanel.BackgroundColor = [0.651 0.651 0.651];
                        app.ControlPanel.FontSize = 14;
        app.ControlPanel.Position = [516 55 260 416];

                             % Create connect_mc
        app.connect_mc = uibutton(app.ControlPanel, 'push');
        app.connect_mc.ButtonPushedFcn = createCallbackFcn(app, @connect_mc_button, true);
          app.connect_mc.BackgroundColor = [1 1 1];
        app.connect_mc.FontColor = [0.149 0.149 0.149];
          app.connect_mc.Position = [50 352 159 22];
            app.connect_mc.Text = 'Connect Microcontroller';

                             % Create charge_caps
        app.charge_caps = uibutton(app.ControlPanel, 'push');
```

```
            app.charge_caps.ButtonPushedFcn = createCallbackFcn(app, @charge_capsButtonPushed, true);
            app.charge_caps.BackgroundColor = [1 1 1];
            app.charge_caps.FontColor = [0.149 0.149 0.149];
            app.charge_caps.Position = [49 290 159 22];
            app.charge_caps.Text = 'Charge Capacitors';

            % Create pulse
            app.pulse = uibutton(app.ControlPanel, 'push');
            app.pulse.ButtonPushedFcn = createCallbackFcn(app, @pulseButtonPushed, true);
            app.pulse.BackgroundColor = [1 0 0];
            app.pulse.FontColor = [0.149 0.149 0.149];
            app.pulse.Position = [50 57 159 82];
            app.pulse.Text = 'Pulse';

            % Create discharge_caps
            app.discharge_caps = uibutton(app.ControlPanel, 'push');
            app.discharge_caps.ButtonPushedFcn = createCallbackFcn(app, @discharge_capsButtonPushed, true);
            app.discharge_caps.BackgroundColor = [1 1 1];
            app.discharge_caps.FontColor = [0.149 0.149 0.149];
            app.discharge_caps.Position = [50 259 159 22];
            app.discharge_caps.Text = 'Discharge Capacitors';

            % Create disconnect_mc
            app.disconnect_mc = uibutton(app.ControlPanel, 'push');
            app.disconnect_mc.ButtonPushedFcn = createCallbackFcn(app, @disconnect_mcButtonPushed, true);
            app.disconnect_mc.BackgroundColor = [1 1 1];
            app.disconnect_mc.FontColor = [0.149 0.149 0.149];
            app.disconnect_mc.Position = [49 321 160 22];
            app.disconnect_mc.Text = 'Disconnect Microcontroller';

            % Create refresh_status
            app.refresh_status = uibutton(app.ControlPanel, 'push');
            app.refresh_status.ButtonPushedFcn = createCallbackFcn(app, @refresh_statusButtonPushed, true);
            app.refresh_status.BackgroundColor = [1 1 1];
            app.refresh_status.FontColor = [0.149 0.149 0.149];
            app.refresh_status.Position = [50 228 159 22];
            app.refresh_status.Text = 'Refresh Status Panel';
        end
    end

    methods (Access = public)

        % Construct app
        function app = TMS
```

```
                    % Create and configure components
                                  createComponents(app)

                  % Register the app with App Designer
                            registerApp(app, app.UIFigure)

            % Execute the startup function
                          runStartupFcn(app, @startupFcn)

                                          if nargout == 0
                                               clear app
                                                     end
                                                     end

                % Code that executes before app deletion
                                      function delete(app)

            % Delete UIFigure when app is deleted
                                     delete(app.UIFigure)
                                                     end
  end
end
```

Microcontroller Code

To avoid having one's eyeballs assaulted, the microcontroller code can be downloaded from: http://sddec18-04.sd.ece.iastate.edu/code/mc_code.txt for viewing.

Paste it into a real editor for insult-free viewing.

```
#............................................................
#include <SoftwareSerial.h>

/*
 * Define constants
 */
#define READ_DELAY 100     //milliseconds delay to ensure next byte is ready to be read from
serial input buffer
```

```
#define MAX_PULSES_ALLOW 10 //max pulses to allow per requirements
#define MIN_PULSES_ALLOW 1 //minimum pulses to allow, per requirements
#define MAX_FREQ_ALLOW 36 //max frequency allowed, per requirements
#define MIN_FREQ_ALLOW 1 //min frequency allowed, per requirements
#define MAX_PWR_LVL_ALLOW 3 //maximum capacitor charge level, per requirements
#define MIN_PWR_LVL_ALLOW 1 //minimum capacitor charge level, per requirements

#define HIGH_TIME_MICROS 500 //high time in microseconds for square wave pulse per
requirements
#define TERMINATOR '\n' //the terminator to be used (if any) for any data received from GUI
#define MAX_COMMAND_SIZE 10 //the maximum length in bytes of a command string that
may be sent from the GUI at one time
#define TIMEOUT 30000 //timeout for capacitor charging in milliseconds (30 seconds). If we
haven't charged by now we have a problem...

#define STARTUP_FREQUENCY 1 // the default frequency that is set when the device is reset
#define STARTUP_PULSE_COUNT 1 // the default pulse count that is set when the device is
reset
#define STARTUP_POWER_LEVEL 1 // the default power level that is set when the device is
reset

#define E_CAP_CHARGE_TIMEOUT 101

/*
 * Digital Output pin for switching main capacitor charging relay on/off.
 */
#define RELAY_PIN 13

/*
 * Digital Output pin to drive IGBT gate via op-amp. This is the main pulse pin.
 */
#define PULSE_PIN 12

/*
 * Analog Input pin for sensing raw ADC value. This ADC value indicates capacitor charge
level.
 */
#define CAP_SENSE_LVL_PIN 5 //analog in pin from which to read capacitor charge level
```

```
/*
 * Bluetooth IO pins
 */
#define BLUETOOTH_SERIAL_RX_PIN 8
#define BLUETOOTH_SERIAL_TX_PIN 7

/*
 * These are the raw ADC values that are experimentally determined to be read
 * on the CAP_LVL_SENSE_PIN pin from the capacitor charge level circuit, for each discrete
power level.
 */
#define CAP_SENSE_LVL_1_THRESH 178 //Expected analog raw value given 80 V on
capacitor
#define CAP_SENSE_LVL_2_THRESH 358 //Expected analog raw value given 160 V on
capacitor
#define CAP_SENSE_LVL_3_THRESH 538 //Expected analog raw value given 240 V on
capacitor

/*
 * Define valid command codes that can be recieved from UI
 */
#define CHANGE_FREQUENCY 'f' //change frequency of IGBT pulses
#define CHANGE_PWR_LVL 'l'  //change max capacitor charge level
#define CHANGE_PULSE_COUNT 'n' //change number of IGBT pulses
#define CHARGE_CAPS 'c'  //begin charging capacitors
#define PULSE 'p'  //fire pulse(s)
#define STATUS 's' //report status/state
#define RESET 'r'  //force microcontroller to known state


//Create Bluetooth serial object using
SoftwareSerial BT_serial(BLUETOOTH_SERIAL_RX_PIN,
BLUETOOTH_SERIAL_TX_PIN); //RX | TX

byte power_lvl = 1;
byte frequency = 1;
byte pulse_count = 1;
char command = '0';
```

```
void setup()
{
  pinMode(RELAY_PIN,OUTPUT);
  pinMode(PULSE_PIN,OUTPUT);
  pinMode(4,OUTPUT);
  digitalWrite(4,LOW);
  //Serial.begin(9600); //Set Baud rate to 9600 //DEBUG
  BT_serial.begin(9600);  //Set Baud rate to 9600
}

/*
 * Loop and handle serial commands from the GUI as they arrive
 */
void loop()
{
   if (BT_serial.available() > 0)
   {
                        command = BT_serial.read();
                if (is_valid_command(command))
                                                        {
                                    switch(command)
                                                        {
                                          case PULSE:
         //Serial.println("In pulse case:"); //DEBUG
         //Serial.println(command);
                                                   pulse();
         //Serial.println("Bytes available:");
           //Serial.println(BT_serial.available());
                                                     break;

                                 case CHARGE_CAPS:
         //Serial.println("In charge caps case."); //DEBUG
         //Serial.println(command);
         charge_caps();
         //Serial.println("Available:");
         //Serial.println(BT_serial.available());
                                                     break;
                       case CHANGE_PWR_LVL:
         //Serial.println("In set power level case."); //DEBUG
                        //Serial.println(command);
                                  set_power_level();
         //Serial.println("Power level set to:");
         //Serial.println(power_lvl);
         //Serial.println("Available:");
         //Serial.println(BT_serial.available());
                                                     break;

         case CHANGE_FREQUENCY:
       //Serial.println("In frequency case. Command is:"); //DEBUG
           //Serial.println(command);
           set_frequency();
           //Serial.println("Frequency set to:");
           //Serial.println(frequency);
           //Serial.println("Bytes available:");
           //Serial.println(BT_serial.available());
                                                     break;

                   case CHANGE_PULSE_COUNT:
           //Serial.println("In count case. Command is:"); //DEBUG
           //Serial.println(command);
                                  set_pulse_count();
           //Serial.println("pulse_count set to:");
           //Serial.println(pulse_count);
           //Serial.println("Bytes available:");
           //Serial.println(BT_serial.available());
                                                     break;

                                          case RESET:
                                               reset();
                                                     break;

                                                   default:
                                                     break;

                                              } //end switch
                                     } //end is valid command
```

```
                              } //end serial available
} //end loop

/*
* Sets the local power level to the value recieved from a GUI change event.
* Checks that the level recieved is one of the defined allowed power levels.
*
* Arguments: pwr_lvl
* Returns: 1 if the power level is successfully set to one of the defined allowed power levels, 0
otherwise.
*/
int set_power_level(){
  //Serial.println("In set_power_level rxed_pwr_lvl is:"); //DEBUG
  int rxed_pwr_lvl = get_command_val();
  //Serial.println(rxed_pwr_lvl);
  if((rxed_pwr_lvl >= MIN_PWR_LVL_ALLOW) && (rxed_pwr_lvl <=
MAX_PWR_LVL_ALLOW)) {
                        power_lvl = rxed_pwr_lvl;
    BT_serial.write(power_lvl);
                                          return 1;
  }
  else {
    BT_serial.write(-1);
                                          return 0;
  }
}

/*
 * Sets the global pulse_count variable to the value returned from get_command_val().
 * Checks that the number of pulses returned is within the range specified by requirements, [1,
10].
 *
 * Arguments: none
 *
 * Returns: 1 if the number of pulses is between MIN_PULSES_ALLOW and
MAX_PULSES_ALLOW, inclusive,
 *                                     0 otherwise.
 *
 */
```

```
int set_pulse_count(){
  int rxed_pulses = get_command_val();
  if((rxed_pulses >= MIN_PULSES_ALLOW) && (rxed_pulses <= MAX_PULSES_ALLOW))
{
                         pulse_count = rxed_pulses;
    BT_serial.write(pulse_count);
                                                   return 1;
  }
  else {
    BT_serial.write(-1);
                                                   return 0;
  }
}


/*
 * This function extracts a subtring representing a number value from the command string
received from the GUI.
 * This substring of the entire command string must still be available in the serial input buffer
 * when this function is called.
 *
 * This function will read the next 3 bytes available in the serial input buffer.
 * The first 2 bytes are interpreted as the integer value of the command. The 3rd byte read is
assumed
 * to be the TERMINATOR charcter.
 *
 * The substring extracted should contain the numerical value of the command.
 *
 * Arguments: none
 *
 * Returns: The substring of the command string representing the command numerical value
 */
int get_command_val(){
  int i = 0;
  char command_value[MAX_COMMAND_SIZE] = {0};
  //Serial.println("BT_serial.avaiable:"); //DEBUG
  //Serial.println(BT_serial.available());
  //Serial.println("sizeof:");
  //Serial.println(sizeof(command_value));
  while(BT_serial.available() && (i < (sizeof(command_value)-1))){
        command_value[i] = BT_serial.read();
 if(command_value[i] == TERMINATOR) {
                                            i++;
      command_value[i] = '\0';
                                          break;
                                               }
                                            i++;
  }
  //Serial.println("In get_command_val raw command val is:"); //DEBUG
  //Serial.println(command_value);
  //Serial.println("In get_command_val command val is:");
  //Serial.println(atoi(command_value));
  return atoi(command_value);
}


/*
 * Sets the pulse frequency to the value received from the GUI.
 * Checks that the value is in allowed range.
 *
 * Reports "ok" to GUI if value is in range and set, "error" otherwise.
 */
int set_frequency(){
  byte rxed_freq = get_command_val();
  if((rxed_freq >= MIN_FREQ_ALLOW) && (rxed_freq <= MAX_FREQ_ALLOW)){
                            frequency = rxed_freq;
    BT_serial.write(frequency);
                                        return 1;
  }
  else {
                              BT_serial.write(-1);
                                        return 0;
  }
}

/*
 * Charge capacitors to the level specified by the pwr_level variable.
 *
 * Arguments: pwr_level,
 * Returns an integer that equals the power level that was set in the range [0,3], or -1 on failure.
 */
int charge_caps()
{
  int raw_adc_val = 0;
  //Serial.println(raw_adc_val); //DEBUG
  switch(power_lvl){
    //Serial.println("In charge_caps: power_lvl is:"); //DEBUG
    //Serial.println(power_lvl);
    //Serial.println(analogRead(CAP_SENSE_LVL_PIN));
                                        case 1:
   if(analogRead(CAP_SENSE_LVL_PIN) < CAP_SENSE_LVL_1_THRESH)
                                              {
        digitalWrite(RELAY_PIN, HIGH);
            unsigned long start_time = millis();
        while(analogRead(CAP_SENSE_LVL_PIN) < CAP_SENSE_LVL_1_THRESH) {
          if(millis()-start_time > TIMEOUT){
          digitalWrite(RELAY_PIN, LOW);
          BT_serial.write(E_CAP_CHARGE_TIMEOUT);
                                      return -1;
                                              }
                                              }
        digitalWrite(RELAY_PIN, LOW);
                                              }
                                         break;

                                        case 2:
      if(analogRead(CAP_SENSE_LVL_PIN) < CAP_SENSE_LVL_2_THRESH)
                                              {
        digitalWrite(RELAY_PIN, HIGH);
            unsigned long start_time = millis();
        while(analogRead(CAP_SENSE_LVL_PIN) < CAP_SENSE_LVL_2_THRESH) {
          if(millis()-start_time > TIMEOUT){
          digitalWrite(RELAY_PIN, LOW);
          BT_serial.write(E_CAP_CHARGE_TIMEOUT);
                                      return -1;
                                              }
                                              }
        digitalWrite(RELAY_PIN, LOW);
                                              }
                                          break;

                                          case 3:
      if(analogRead(CAP_SENSE_LVL_PIN) < CAP_SENSE_LVL_3_THRESH)
                                              {
        digitalWrite(RELAY_PIN, HIGH);
           unsigned long start_time = millis();
        while(analogRead(CAP_SENSE_LVL_PIN) < CAP_SENSE_LVL_3_THRESH) {
          if(millis()-start_time > TIMEOUT){
           digitalWrite(RELAY_PIN, LOW);
        BT_serial.write(E_CAP_CHARGE_TIMEOUT);
                                       return -1;
                                              }
                                              }
        digitalWrite(RELAY_PIN, LOW);
                                              }
                                          break;

                                        default:
                                          break;
  }
  //Serial.println(raw_adc_val); //DEBUG
  BT_serial.write(power_lvl);
  return power_lvl;
}

/*
 * This function checks the global status of the device, then fires pulses to drive the IGBT
 * based on device configuration.
 *
 * Arguments: none
 * Returns: 1 if the device is ready and pulses were fired successfully. Returns 0 otherwise.
 */
int pulse(){

  short freq_to_lowtime_map[36] =
  {995,500,333,245,200,166,143,122,111,100,91,83,77,71,66,62,
  59,55,52,49,47,45,43,41,40,38,37,36,34,33,32,31,30,29,28,27};

  short lowtime = freq_to_lowtime_map[frequency-1];
  byte pulse_cntr = 0;
  while (pulse_cntr < pulse_count) {
    digitalWrite(PULSE_PIN, HIGH);
    delayMicroseconds(HIGH_TIME_MICROS);
    digitalWrite(PULSE_PIN, LOW);
                                    delay(lowtime);
                                                  /*
    digitalWrite(4,HIGH); //DEBUG
                                          delay(500);
    digitalWrite(4,LOW);
                                    delay(lowtime);
                                                  */
                                      pulse_cntr++;
  }
  return 1;
}

/*
 * Send status to Matlab
 *
 * Arguments: none
 * Returns:
 */
int report_status(){
  if(BT_serial.isListening()){
    BT_serial.write(power_lvl);
    BT_serial.write(pulse_count);
    BT_serial.write(digitalRead(RELAY_PIN));
  }
  return 0;
}

/*
 * Closes main power relay and resets state to startup values.
 *
 * Arguments: none
 * Returns: 0 on success
 */
int reset(){
  digitalWrite(RELAY_PIN,LOW);
  digitalWrite(PULSE_PIN,LOW);
  frequency = STARTUP_FREQUENCY;
  pulse_count = STARTUP_PULSE_COUNT;
  power_lvl = STARTUP_POWER_LEVEL;
  BT_serial.write(1); //send 1 to GUI to signal successful reset
  return 1;
}

/*
 * Verifies that the given command is one of the defined valid commands.
 *
 * Arguments: command
 * Returns: 1 if the command is valid, 0 if not valid.
 */
int is_valid_command(char command){
  //Serial.println("In is_valid_command:"); //DEBUG
  //Serial.println("Command is:");
  //Serial.println(command);

  char valid_cmds[] = {'l', 'n', 'c', 'p', 's', 'r', 'f'};
  //Serial.println("sizeof is:"); //DEBUG
  //Serial.println(sizeof(valid_cmds));
  int i = 0;
  for(i=0; i<sizeof(valid_cmds); i++){
    //Serial.println(valid_cmds[i]); //DEBUG
		if(command == valid_cmds[i]){
			return 1;
		}
  }
  return 0;
}
```

5.6 References

[1] Garcia, N. (2017). "Transcranial Magnetic Stimulation- TMS." Internet: Available: http://www.neuromodulation.com/TMS, April 25, 2017 [April 20, 2018].

[2] M. Lu and S. Ueno. (2017). "Comparison of the Induced Fields using Different Coil Configurations During Deep Transcranial Magnetic Stimulation." *PLOS One.* [Online]. 12(6), p.e0178422. [April 20, 2018].

[3] Y. Roth, G. Pell, and A. Zangen. (2013). "Commentary on: Deng et al., Electric Field Depth-Focality Tradeoff in Transcranial Magnetic Stimulation: Simulation Comparison of 50 Coil Designs." *Brain Stimulation*. [Online]. 6(1), pp. 14-15.

[4] S. Ulven, et al. "TMS: Transcranial Magnetic Stimulation.", unpublished.

[5] G. Bulleit, et al. "High Current Pulse Generator.", unpublished.

[6] M.J.R. Polson. "Magnetic Stimulator for Neuro-Muscular Tissue." U.S. Patent 5 766 124, Jun. 16, 1998.

[7] J. Selvaraj, et al. "TMS: Design of a Stimulator and Focused Coil for the Application of Small Animals.", *IEEE Transactions on Magnetics,* vol. 54, no. 11, pp. 1-5, Nov. 2018.

[8] *IEEE Standard for Digital Recorders for Measurements in High-Voltage Impulse Tests* IEEE Std 1122-1998, 1998.