ENGINEERING

McMaster University

MECHTRON 3K04: Software Development PACEMAKER Project (Group 19) Documentation

Himanshu Singh (singh41, 400377922)

Mathew Galuszka (galuszkm, 40036788)

Shaan Suthar (suthas2, 400401752)

Shivan Gaur (gaurs5, 400393966)

Varun Kothandaraman (kothandv, 400396038)

Christopher Nazarian (nazariac, 400388931)

Table of Contents

Revision History

Revision	*Description*	*Date (MM/DD/YYYY)*
Assignment 1	Assignment 1 Pacemaker design, Device Controller Monitor (DCM) and Design Documentation Submission. **Authors:** Himanshu Singh, Mathew Galuszka, Shaan Suthar, Shivan Gaur, Varun Kothandaraman, Christopher Nazarian	10/15/2023
Assignment 2	Assignment 2 Pacemaker design, Assurance Case, Device Controller Monitor (DCM), Design Documentation **Authors:** Himanshu Singh, Mathew Galuszka, Shaan Suthar, Shivan Gaur, Varun Kothandaraman, Christopher Nazarian	12/03/2023

Introduction

In the realm of cardiac healthcare, artificial pacemakers stand as pivotal devices, engineered to meticulously regulate the heart's rhythm. These sophisticated electrical units are strategically implanted to interface directly with the heart via the subclavian vein. Their primary function is to deliver precise electrical pulses to the right atrium and ventricle, thereby rectifying irregular heartbeats. This intervention is crucial for stabilizing the cardiac conduction system, ensuring seamless blood circulation, and, ultimately, preserving the patient's overall health and wellbeing.

This document delves into the intricate journey of our pacemaker's genesis - from conceptual design to its final, functional form. Central to our development process was the utilization of MATLAB Simulink, a dynamic simulation environment that allowed us to meticulously model and refine our pacemaker's architecture. Complementing Simulink, we employed Heartview, a sophisticated cardiac simulation tool, to rigorously test and validate the functionality of our state flow diagrams. This dual-platform approach was instrumental in achieving a high degree of accuracy and reliability in our design.

A cornerstone of our pacemaker's innovation is the incorporation of the Device Controller Monitor (DCM). The DCM is ingeniously designed to not only facilitate adaptive responses to the nuanced variances in each patient's cardiac cycle but also to provide a user-friendly interface for monitoring and modifying the pacemaker's operational parameters. This feature is particularly vital for tailoring the device to the unique cardiac profiles of individual patients.

Moreover, our DCM is equipped with an advanced data management system capable of registering and storing information for up to ten users. This multi-user functionality is not only a testament to the pacemaker's versatility but also enhances its practicality in diverse clinical settings. Each registered user has the autonomy to customize the default settings of the pacemaker, allowing for personalized cardiac care.

In essence, this project encapsulates a blend of cutting-edge engineering and patient-centered design, aiming to set a new benchmark in cardiac pacemaker technology. It truly incorporates the essence of mechatronics and embedded systems, with a strong emphasis on why software development and safety is so essential. The following sections will provide an in-depth exploration of each phase of our journey - from the initial design blueprints to the meticulous simulation trials, leading up to the real-world implementation of our state-of-the-art pacemaker.

Planning & Research

Preliminary

Before beginning with either the DCM or Simulink, a GitHub repository was created to allow for access and collaboration on files between group members. It was decided that splitting into two subgroups-one to handle the Simulink and Pacemaker and the other the DCM-would be the most efficient way of tackling both components in the given time frame. This project emphasized understanding the requirements and accounting for a wide variety of test cases to ensure safety. As such, all the critical documentation, including the previous assignment, assignment two requirements, the pacemaker shield and main pacemaker documents were reviewed prior to the start of any physical work.

Pacemaker

To help with debugging, a decision was made to create each pacing mode in its own MATLAB Simulink file, and then combine all of the modes into one single file when they were all functioning correctly. In retrospect, this also helped with compile times and overall organization of code for testing.

Device Controller Monitor (DCM)

Due to the flexibility in the DCM Requirement Specifications regarding the selection of technologies, it is imperative to thoroughly evaluate all available options before making a decision on the implementation. To facilitate this technology selection process, the DCM selection has been categorized into three distinct areas: GUI display, data management, and serial communication. Each category's potential implementations were scrutinized and assessed to ascertain their suitability for the project.

DCM Technologies Decision Comparison

DCM Comparison – GUI Breakdown

Python →Tkinter: The Tkinter library in Python is an excellent choice for developing a straightforward, locally run pop-up application, particularly when the project's interface requirements are not overly complex. Additionally, Python offers another significant advantage for this project due to the availability of a well-documented library for facilitating serial communication between the pacemaker and Python, ensuring smooth and efficient data exchange.

Furthermore, opting for Python provides the added benefit of capitalizing on the collective expertise of the project team. Every team member is guaranteed to have experience in programming with Python, as it was introduced during their first year in the Engineering program or the Integrated Biomedical Engineering and Health Sciences (IBioMed) program. This shared proficiency in Python will streamline collaboration and development, making it an ideal choice for our project.

Java→Swing: Within our team of six members, two possess a wealth of experience in developing intricate video games and applications using the Java Swing library. Leveraging Java for our project offers several compelling advantages. Firstly, it allows us to harness its robust object-oriented capabilities, which can greatly enhance the development process. Additionally, Java follows the Model-View-Controller (MVC) architecture, which empowers us to create a more adaptable and flexible User Interface (UI).

However, it's essential to acknowledge a potential drawback. Our Teaching Assistants (TAs) lack familiarity with serial communication in Java, a technical aspect that may not be directly required for Assignment 1 but holds future significance. Navigating the intricacies of a syntax-heavy language like Java without the guidance of teaching faculty could pose challenges down the road. Thus, while Java offers notable advantages, we must consider the potential hurdles of limited TA support when making our decision.

HTML/CSS/JavaScript: The most widely recognized and preferred languages for developing frontend applications are the cornerstone web development languages: HTML, CSS, and JavaScript. Remarkably, three out of our six group members possess prior experience working with these frameworks. This foundation in these languages bestows upon us a considerable advantage.

Nevertheless, it's crucial to acknowledge a potential challenge. One significant drawback of relying on HTML, CSS, and JavaScript is ensuring that our Teaching Assistants (TAs) can comprehensively understand our code. This challenge primarily stems from the intricacies of commenting CSS code, which can often be perplexing and problematic. As such, while these web development languages offer substantial benefits, it's essential to be mindful of the potential complexity in code comprehension for our TAs.

Decision:

In our pursuit of Graphical User Interface (GUI) development, our team arrived at the deliberate choice of harnessing the power of TKinter in tandem with the Python programming

language. This decision was underpinned by a multitude of compelling factors, as previously deliberated. Firstly, TKinter emerged as the ideal candidate for crafting streamlined, locally executed pop-up applications, perfectly aligning with the simplicity our project required, sparing us from the complexities of an overly intricate interface. Moreover, Python's capability to do object-oriented programming and simplicity made it the most optimal choice for the DCM.

What truly swayed our decision, however, was the prospect of future-proofing our serial communication capabilities, thus alleviating a considerable burden from our workload. The realization that we wouldn't need to search externally for these capabilities in the future proved to be the decisive factor. While we acknowledged the absence of bonus points for extravagant GUI implementations, we opted for an aesthetic and functional GUI that exceeded the minimum requirements but stayed within reasonable bounds, at least for this initial assignment. This prudent approach ensured that our shared expertise, honed during our first-year experiences in Engineering and IBioMed programs, paved the way for a seamless collaborative development process. In sum, the amalgamation of these factors fortified our unwavering choice of TKinter and Python as our preferred technologies for GUI development.

DCM Comparison – Data Management

For our implementation, we considered various technologies and languages.

MongoDB: We could implement MongoDB using Python with MongoPy or JavaScript with Mongoose. MongoDB is a non-relational database known for its excellent documentation and relatively easy learning curve. It also offers supporting packages that make integration with Flask applications more convenient. However, the development team's experience with MongoDB was somewhat limited.

Firebase: We could implement Firebase using JavaScript with the Firebase SDK. Firebase is another non-relational database, known for its excellent documentation. However, it has a requirement that users must register an account with Google Apps and provide payment information, which could be a drawback for some.

SQL: For SQL databases, we would use Postgres and Sqlite3 with Python. SQL databases are relational, which aligns well with the data representation needs of our project. Using Sqlite3 with Python allows for direct integration into a Python project. As a result, SQL databases received a relatively high overall score of 7 out of 10, making them a strong contender for our implementation.

Locally Stored: An alternative approach, distinct from relying on an online data management technology, involves the local storage of all data and states within the computer running the DCM file. One method to achieve this is by saving the user's data in a text file on the local machine. While some may initially view this choice as a potential manifestation of laziness, the rationale behind it is, in fact, far more grounded in logic than it may seem. This decision is particularly pertinent given the nature of the project, which revolves around a pacemaker.

In the context of a pacemaker, mandating an internet connection for updating its settings can introduce significant risks in real-life scenarios. Therefore, it becomes considerably safer and more prudent to leverage the concept of encapsulation, wherein the data and the program coexist locally. This approach minimizes potential vulnerabilities associated with external dependencies, ensuring that the pacemaker's functionality remains robust and reliable, free from the inherent uncertainties of online connectivity.

Decision: Indeed, when we delve deeper into the considerations outlined earlier, the choice of local storage for data management becomes not only reasonable but also the most sensible course of action. The paramount reason, as previously discussed, is the inherent safety and reliability it offers in the context of a pacemaker project.

As a life-critical medical device, the pacemaker's operation demands an extra layer of caution. Requiring an internet connection to update its settings would introduce a grave level of risk into real-life scenarios. Connectivity issues, data breaches, or even the possibility of unauthorized access could jeopardize the pacemaker's functionality, potentially endangering the patient's life.

By opting for local data storage, we adhere to the principles of encapsulation, ensuring that the data and program operate in a self-contained environment, impervious to external disruptions. This approach is not a manifestation of laziness, as some might initially perceive it,

but rather a strategic choice made to safeguard the integrity of the pacemaker and prioritize patient safety above all else.

Additionally, the usernames and passwords are encoded rather than stored as plain text. The password is hashed using the SHA-256 algorithm. This is a widely used and secure cryptographic hash function. The hashed password is then stored, enhancing the security of user credentials within the system. It's important to note that when users log in, their entered passwords are hashed using the same algorithm, and the resulting hash is compared to the stored hash for authentication. This approach ensures that the actual password is never stored or transmitted in an easily readable format. Therefore, by following this industry security best practice we can protect the patients' sensitive, personal, and private information. Regarding the specific filetype, we originally used .txt text files to store the user data, but we later opted for JSON files due to having better manipulability.

In essence, when the project's unique circumstances and life-critical implications are factored in, the preference for local storage emerges as the most rational and responsible choice for data management.

DCM Comparison – Serial Communication

In terms of serial communication, this is truly where Python stood out among other technologies. PySerial is a Python module that encapsulates the access for the serial port. Its comprehensive documentation and widespread use in the programming community made it an attractive choice for our needs.

Python's popularity and the rich resources available for PySerial provide a significant advantage. The abundant examples, tutorials, and community support make troubleshooting and learning more accessible and efficient. This aspect is particularly beneficial for a project where time and resources are limited, and the learning curve needs to be as gentle as possible.

Another critical factor in favor of Python and PySerial is the compatibility with various operating systems, including Windows, macOS, and Linux. This cross-platform compatibility ensures that our application can be used seamlessly across different environments, which is essential for a project that might be deployed in various clinical settings with different types of computer systems.

The decision to use Python and PySerial was further reinforced by their robust capabilities in handling serial communication. PySerial provides a straightforward and effective way to read from and write to serial ports, which is a fundamental requirement for our pacemaker project. Its ability to handle different types of serial data and its reliability in maintaining stable connections are key aspects that align well with the critical nature of our project.

In conclusion, the combination of Python with its PySerial library stood out as the best choice for serial communication.t. The wealth of documentation, ease of use, cross-platform compatibility, and the reliable handling of serial data were the pivotal factors that influenced our decision. This choice ensures that we can develop a robust and effective solution that meets the stringent requirements of a medical device like a pacemaker, prioritizing patient safety and device reliability.

Requirements & Specifications

Previous Pacemaker Requirements:

For a pacemaker in permanent state, implement stateflows in Matlab Simulink for the following pacemaker modes: AOO (atrium paced, no chambers sensed, no response to sensing), VOO (ventricle paced, no chambers sensed, no response to sensing), AAI (atrium paced, atrium sensed, inhibition in response to sensing), VVI (ventricle paced, ventricle sensed, inhibition in response to sensing). The parameters required for a stateflow should be programmable and are listed below according to mode:

Parameter	AOO	VOO	AAI	VVI
Lower rate limit	X	X	X	X
Upper rate limit	X	X	X	X
Atrial amplitude	X		X	
Ventricular amplitude		X		X
Atrial pulse width	X		X	
Ventricular pulse width		X		X
Atrial sensitivity			X	
Ventricular sensitivity				X
VRP				X
ARP			X	

Table 1: Parameters for Pacemaker modes

Hardware hiding is also required so that even if the pin map being used were to be altered, the stateflows would not change.

New Pacemaker Requirements:

Using the model developed in the previous assignment, we are required to implement four additional modes to the Matlab Simulink stateflow. Listed in order: AOOR (rate adaptive pacing of the atrium with no sensing and no response to sensing), VOOR (rate adaptive pacing of the ventricle with no sensing and no response to sensing), AAIR (Rate adaptive pacing of the atrium with sensing for natural heart beats and inhibition of pacing according to sensed beats), VVIR (Rate adaptive pacing of the ventricle with sensing for natural heart beats and inhibition of pacing according to sensed beats). Common between these four new modes is the requirement for rate adative pacing which requires the pulse rate to increase based on patient activity sensed by the pacemaker's on board accelerometer. With these new modes comes new required parameters specific to rate adaptive pacing. These include: activity threshold, reaction time, recovery time, response factor, and maximum sensor-driven rate. In addition to the new rate adaptive parameters, atrial-ventricular delay (AVDelay) and post ventricular atrial refractory period (PVARP) must be implemented. The programmable parameters are listed below by mode:

Parameter	AOO	VOO	AAI	VVI	AOOR	VOOR	AAIR	VVIR
Lower rate limit	X	X	X	X	X	X	X	X
Upper rate limit	X	X	X	X	X	X	X	X
MSR					X	X	X	X
AVDelay								
Atrial amplitude	X		X		X		X	
Ventricular amplitude		X		X		X		X
Atrial pulse width	X		X		X		X	
Ventricular pulse width		X		X		X		X
Atrial sensitivity			X				X	
Ventricular sensitivity				X				X
VRP				X				X
ARP			X				X	
PVARP			X				X	
Activity Threshold					X	X	X	X
Reaction Time					X	X	X	X
Response Factor					X	X	X	X
Recovery Time					X	X	X	X

Table 2: Parameters for Pacemaker modes

All of these new modes should be implemented in a single model. Once built onto the board, the pacemaker should also contain functionality that allows it to use serial communication with the DCM for the sending and receiving of information. This is required for delivering egram data to the DCM, receiving and setting the programmable parameters used in the pacemaker's stateflow, and sending back confirmation they were received correctly.

DCM Requirements

Before we began designing our DCM, we first outlined the objective and purpose of each component interacting with our pacemaker. All of these requirements are outlined below in respect to their importance and purpose:

Login Screen

Objective: Develop a secure and user-friendly login system that allows for account creation, access, and secure data storage of up to 10 users.
The system should support three primary functionalities: Logging in, creating an account, and securely storing user information. The system should provide feedback to the user regarding login success or failure upon interacting with UI elements as there should be clear error messages for common issues like incorrect passwords, empty usernames or passwords, and already existing usernames. The number of maximum creatable accounts should be 10.

Account Creation

1. There should be a clearly labelled option or button to create a new account.

2. During account creation, users should input a username and password, and be required to retype the password.
3. The initial password and the retyped password must match.
4. Usernames should be unique; the system must notify the user if the chosen username already exists.
5. Provide feedback upon successful account creation or if there are any issues.
6. No more than 10 accounts should be allowed to be created; the system must notify the user if the maximum number of accounts is reached.
7. There should be a button to go back to the home screen.

Login Functionality:

1. The interface should have fields for the user to input their username and password.
2. Upon entering the correct username and password, users should be redirected to the next stage or main application interface.
3. If the username or password is incorrect, a clear notification should be displayed indicating the error.
4. Users should be given the option to reset their password or create a new account if they can't log in.
5. Additionally, the screen should display the time to the user.
6. There should be a button to go back to the home screen.

Secure Data Storage

1. User data storage should ensure minimal access while maintaining functionality.
2. Data should be stored in a "pacemaker" (assuming this is a secure storage option in the context given), which allows for continuous access regardless of external conditions.
3. All user data, especially passwords, should be encrypted using a strong and modern encryption algorithm.

Main App Interface

Objective: Design a user-friendly interface that allows users to navigate pacemaker modes, view and select parameters, and differentiate between regular and admin accesses. Additionally, the main app should have the ability to start and stop the program.

User Account Navigation:

1. Users must have the capability to log out and be redirected back to the login screen.

User Interface (UI) Functionality:

1. The UI should be clear to use and not cause confusion from the user.

Parameters Display:

1. The system should display eight distinct pacemaker modes: AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR.
2. Each mode should be accompanied by its related parameters as referenced in the "PACEMAKER Document: Table 6".

Mode Selection and Parameter Preview:

1. Users should have the option to select any of the pacemaker modes presented.
2. For every selected mode, users should be able to edit the corresponding parameters in the "PACEMAKER Documentation: Table 7"

Parameter Default Settings:

1. On accessing a pacemaker mode for the first time or after a reset, users should see the nominal values set for each parameter.

Additional Considerations:

1. Implement a clear hierarchy in displaying the pacemaker modes and their associated parameters to avoid user confusion.
2. Prevent buttons from being pressed in an illogical sequence and signify this to the user. For example, the user shouldn't be able to press "Stop" unless "Run" been clicked so grey it out.
3. Implement UI theme changes for accessibility in reading, as in light and dark mode

Design

Pacemaker Design

As mentioned in Planning and Research, the development of each mode occurred within individual files before being implemented. Due to the nature of how the pacing system operates, all modes share a dual state system where the only differences are how the transitions occur. These two states are called Pace and Refractory, and the following defines how these states operate in AOO and AAI.

In Refractory, Capacitor C22 needs to be charged and Capacitor C21 needs to be discharged. Because these operations are mutually exclusive, they were incorporated into a single state. According to the mandatory pin arrangement provided in the documentation, discharging C21 is accomplished by setting PACE_GND_CTRL and ATR_GND_CTRL high. Charging C22 is accomplished by setting PACE_CHARGE_CTRL high and setting PACING_REF_PWM to Atrium_Amplitude, which is the charging duty cycle. How the duty cycle is calculated is outlined in Hardware Hiding. In Pace, ATR_PACE_CTRL and PACE_GND CTRL are set high to apply the pace to the heart. For VOO and VVI, the same variables are used with their names switched from atrium to ventricle. These changes include swapping ATR_GND CTRL to VENT_GND CTRL, Atrium_Amplitude to Ventricle_Amplitude, ATR_PACE_CTRL to VENT_PACE_CTRL, and Atrium_Pace_Width to Ventricle_Pace_Width. Additionally, during the pace state the LED on the pacemaker is set high to visualize the pacing of the heart. During the refractory state it is set low.

The following section details the transitions of each individual mode.

AOO/AOOR

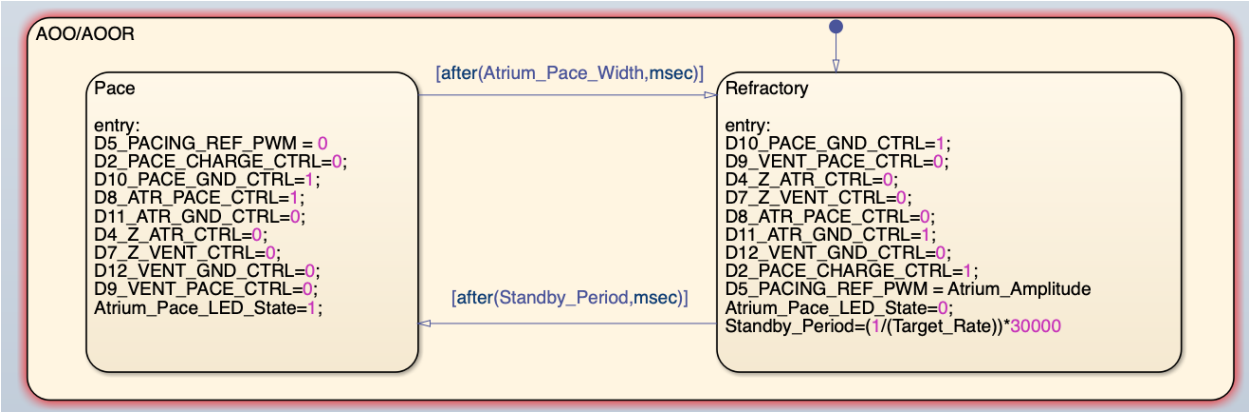


Figure 1. AOO Stateflow Diagram

After all of the output pins are set in Refractory, the standby period is calculated by converting the Lower Rate Limit from BPM to a period in milliseconds. As this is a non-sensing mode, the standby period is determined solely by the LRL. The transition to Pace occurs after the calculated standby period. After all the output pins are set in Pace, the transition to refractory

occurs after the Atrium_Pace_Width, which is a programmable parameter that defines the pulse width of the pace sent to the heart.

VOO/VOOR

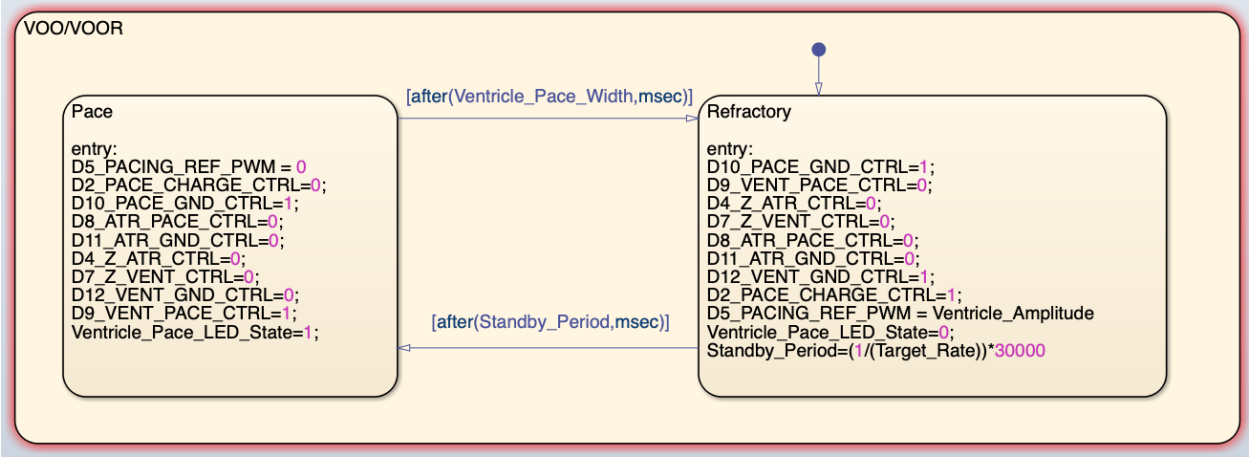


Figure 2. VOO Stateflow Diagram

VOO is very similar to AOO described above, only swapping Atrium_Pace_Width to Ventricle_Pace_Width.

AAI/AAIR

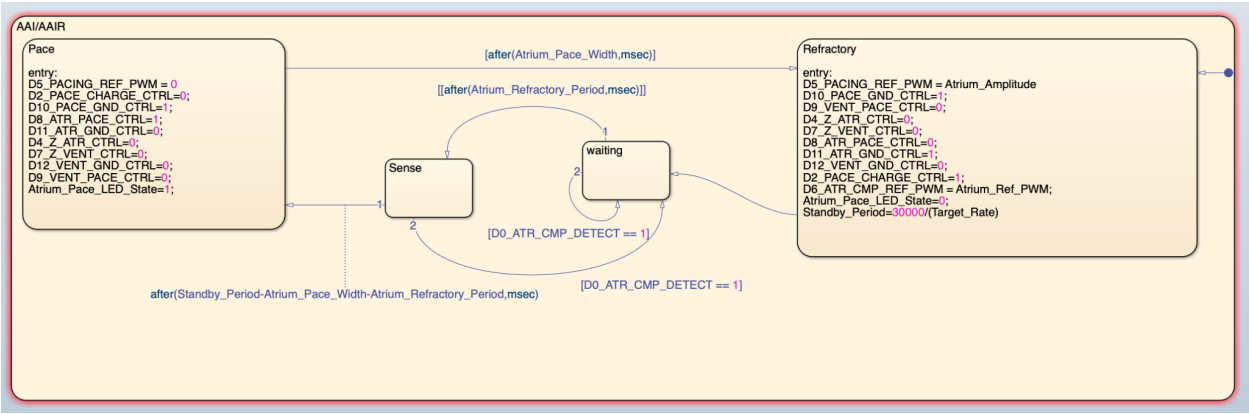


Figure 3. AAI Stateflow Diagram

After all the output pins are set in Refractory, the state begins waiting for the refractory period to expire before moving onto the standby period. If it senses a natural pulse, it routes back to the beginning of the refractory state and resets the refractory period. If not, the state moves to the next junction where it begins waiting out the standby period. The standby period is calculated the same as AOO/VOO, but refractory period is subtracted from Standby to account for the time used up by the initial “after” statement. Like before, if it senses a natural pace, it returns to the beginning of Refractory and restarts the entire process, and if it does not sense a natural pace, the stateflow transitions to the Pace state. After all the output pins are set in Pace, the transition to refractory occurs after the Atrium_Pace_Width, which is a programmable parameter that defines the pulse width of the pace sent to the heart.

VVI/VVIR

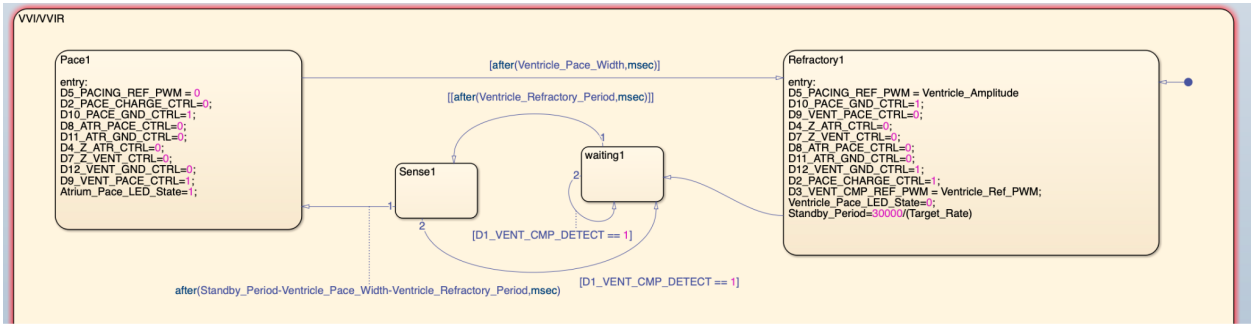


Figure 4. VII Stateflow Diagram

Like VOO to AOO, VVI is very similar to AAI, where the only difference is changing all the atrium variables to ventricle variables.

Mode Switching

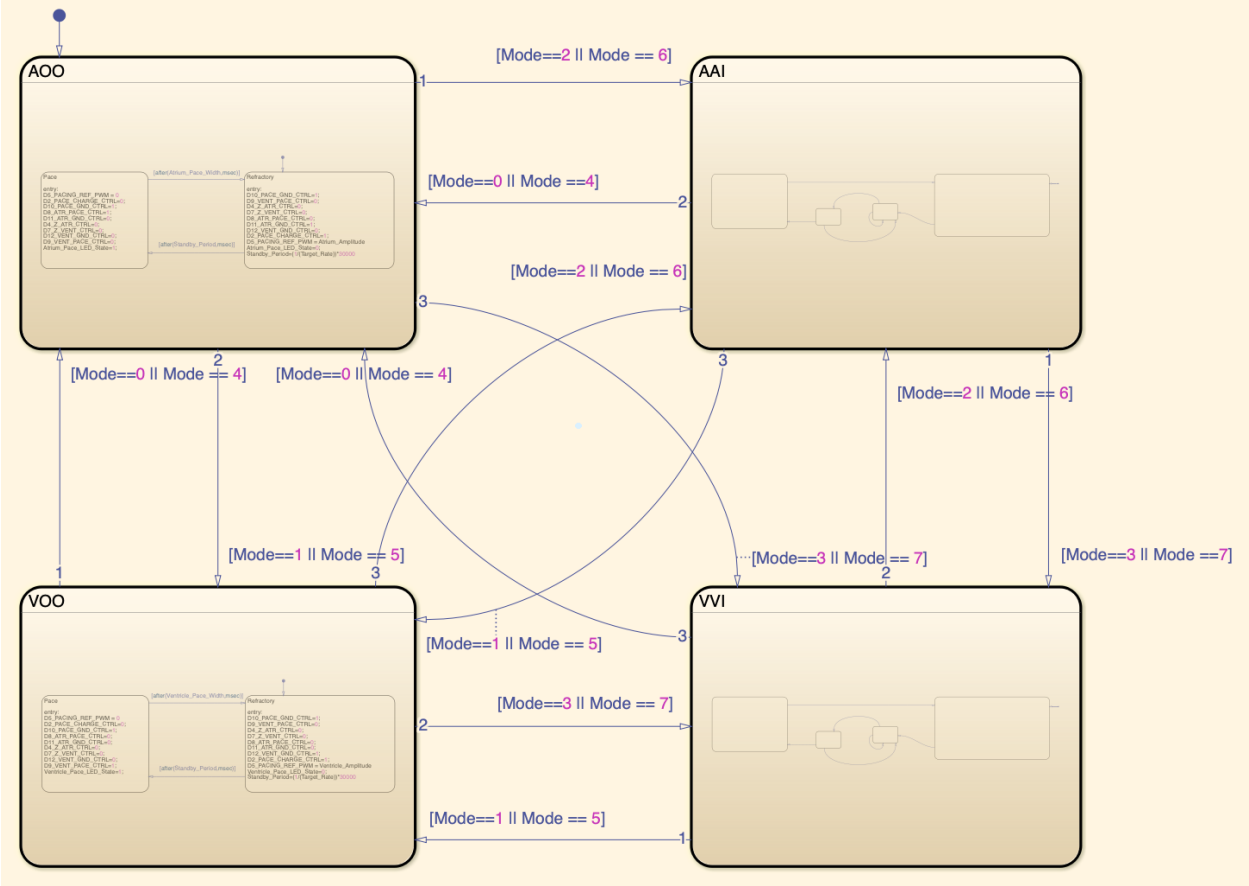


Figure 5. Mode Switching Logic Diagram

Mode switching is accomplished by associating integer values to the various states as follows; AOO=0, VOO=1, AAI=2, VVI=3, AOOR = 4, VOOR= 5, AAIR = 6, VVIR = 7. Depending on the active state and the value of Mode, the stateflow follows the transition from the current state to the desired state.

Hardware Hiding

To ensure hardware hiding between the stateflow and the board and parameters, subsystems were created for hardware outputs and input parameters. Inside of the subsystems, inports were labelled and then routed to their corresponding pin out. In doing so the outputs/inputs of the stateflow could be routed through the inports so that they were not directly touching any digital outputs or inputs. On the input side logic blocks were used to hide any duty cycle conversions which are dependent on-board specific voltages.

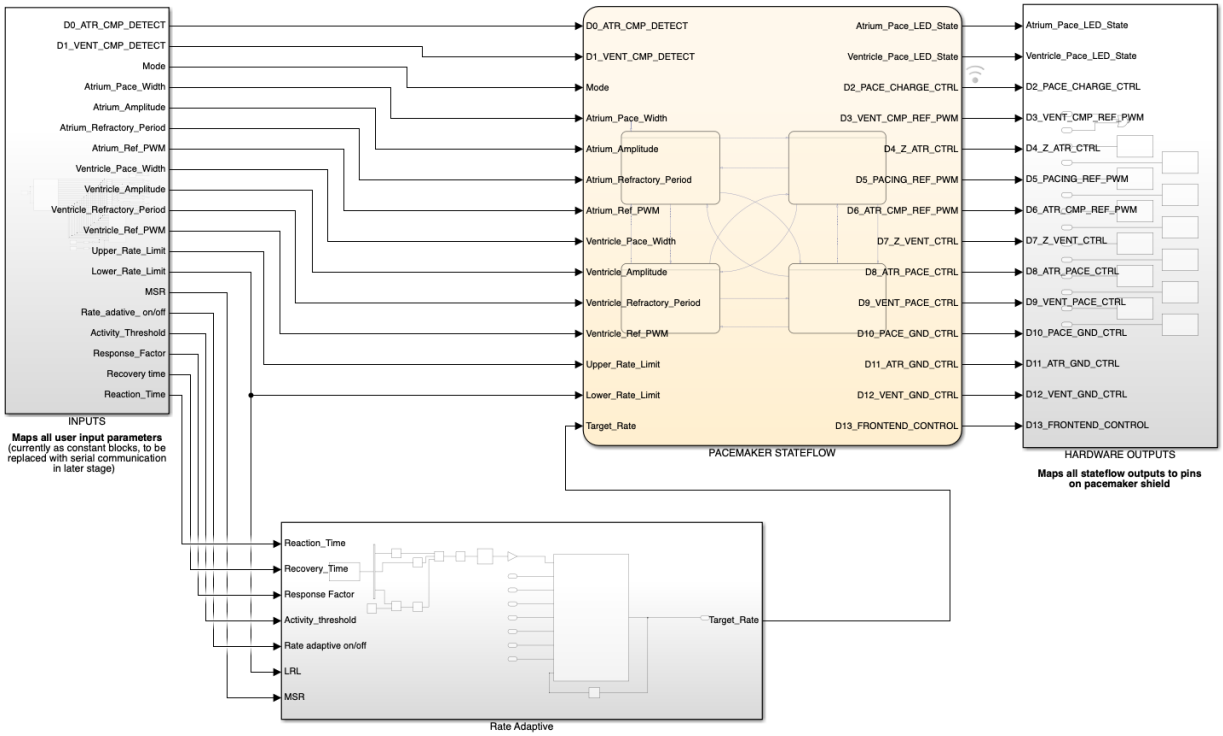


Figure 6. Hardware Hiding Sub-Systems Diagram

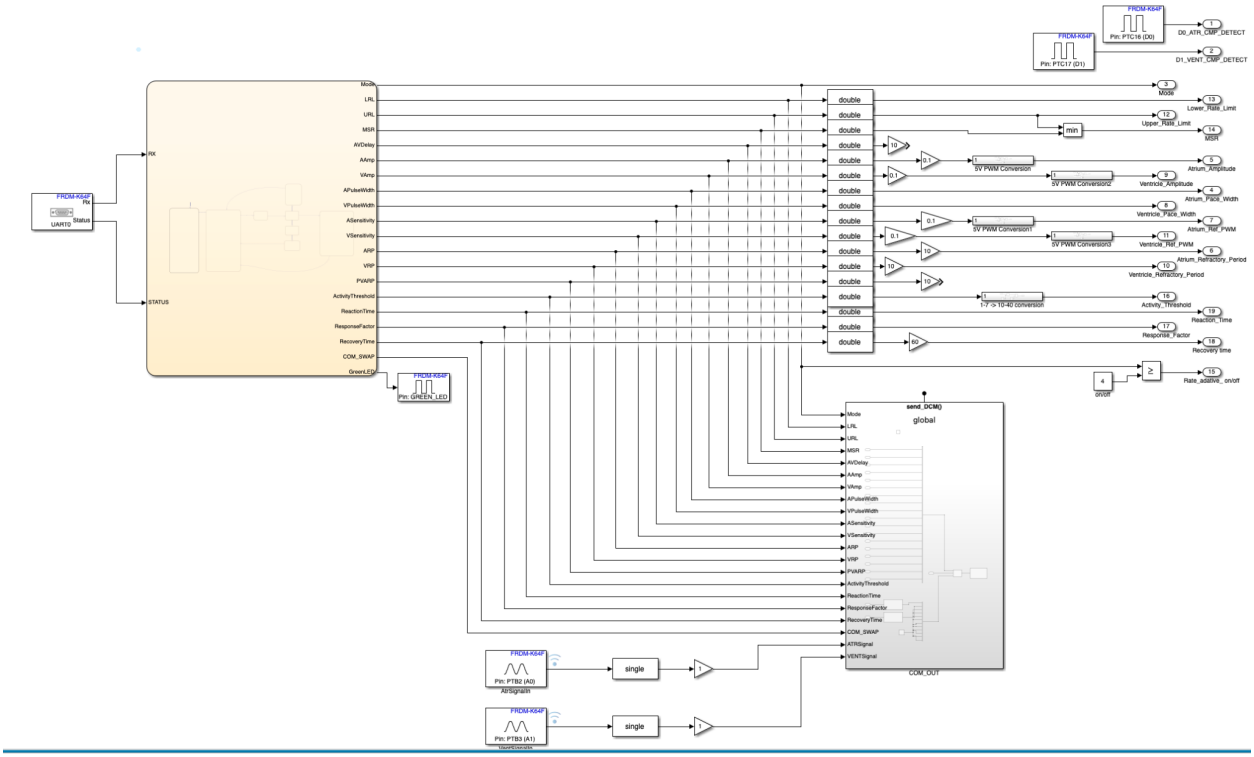


Figure 7. Input Parameters Diagram

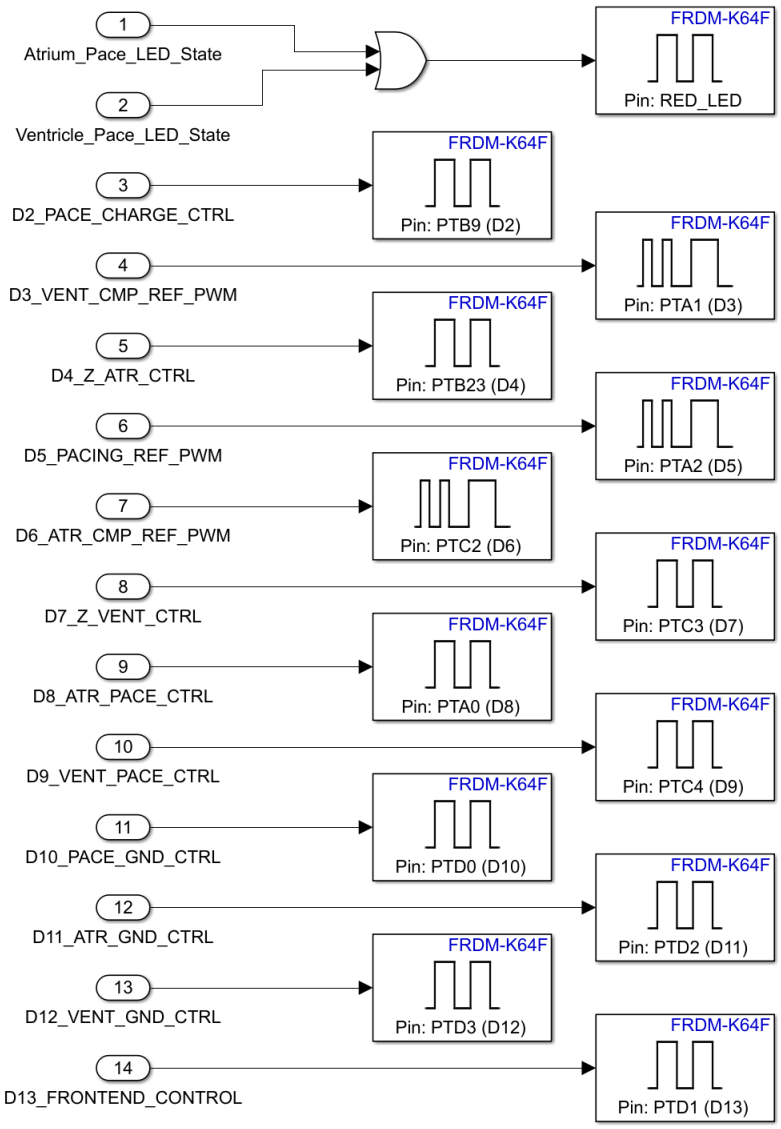


Figure 8. Hardware Outputs Diagram

Serial Communication

As mentioned in the requirements, the pacemaker must be controlled solely by the DCM using serial communication, specifically UART. Communication is accomplished in three different modes, send, receive, and egram mode.

It is important that the byte pack was defined prior to developing the stateflow within Matlab. This is so that the SYNC, FUNCTION, and variable order is known so that it can be parsed within the stateflow. In order to achieve the shortest byte pack, all values are converted to uint8 within matlab, and reconverted back to their original data types within python. This was a design decision made so that data transfer would be faster and more efficient. The byte pack is defined

as follows:

1. SYNC - Must be 22 to acknowledge message
2. FUNCTION - 34 for send, 85 for receive, 56 for egram
3. Mode
4. LRL
5. URL
6. MSR
7. AVDelay
8. AAmp
9. VAmp
10. APulseWidth
11. VPulseWidth
12. ASensitivity
13. VSensitivity
14. ARP
15. VRP
16. PVARP
17. ActivityThreshold
18. ReactionTime
19. ResponseFactor
20. RecoveryTime

Since all values are converted to integers (with respect to their minimum and maximum values), all data types are sent as uint8, and the total pack data length is **20 bytes.** If other data types such as single or uint16 were to be used, the pack would be much larger. The process of parsing through this byte pack is as shown in the stateflow below.

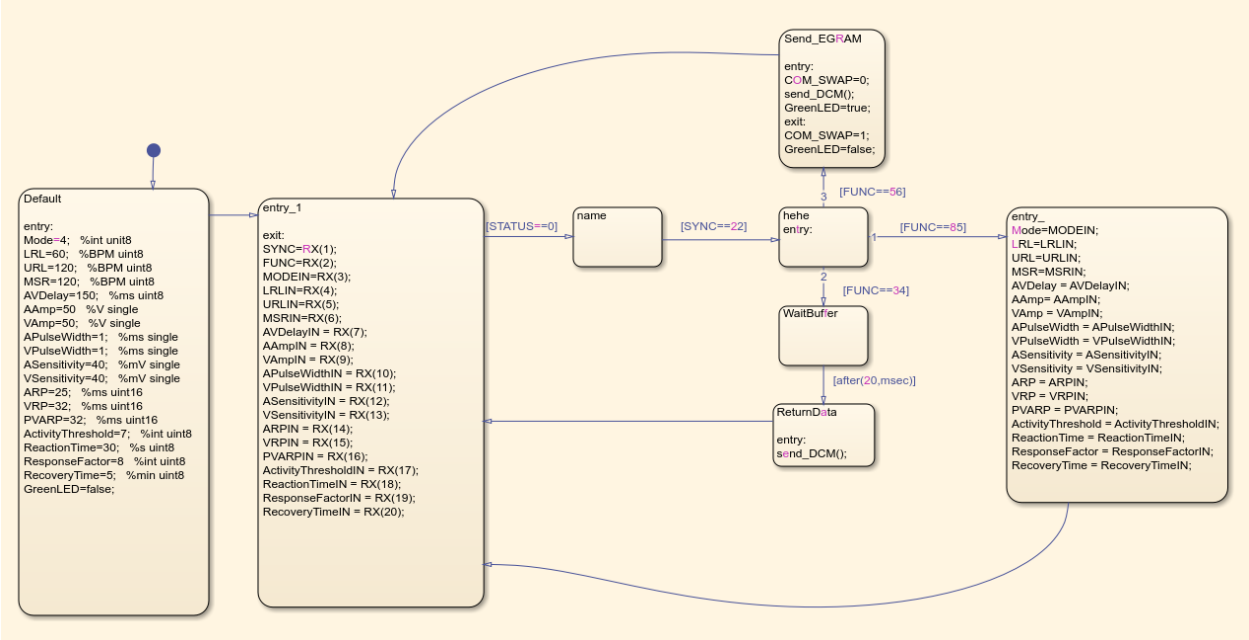


Figure 9.

The beginning default state sets an initial value to all variables. These values are arbitrary, as they are to be replaced further into the stateflow. First, the entry_1 state indexes the byte pack and extracts all of the values. Once the status and sync conditions are met, the stateflow checks the FUNCTION variable to see which communication mode (send/receive/egram) is selected. If the pack indicates receive mode, all of the variables extracted from the pack replace the pacemakers variables. If the pack indicates send mode or egram mode, the external send_DCM() function is called, which has the capabilities to send either the current values in the pacemaker, or the egram values of the pacemaker.

Send_DCM() function

As mentioned above, the send DCM function can send either the current settings within the pacemaker, or send egram data back to the DCM. Toggling back and forth within these two functionalities is done with MATLAB's switch block like so:

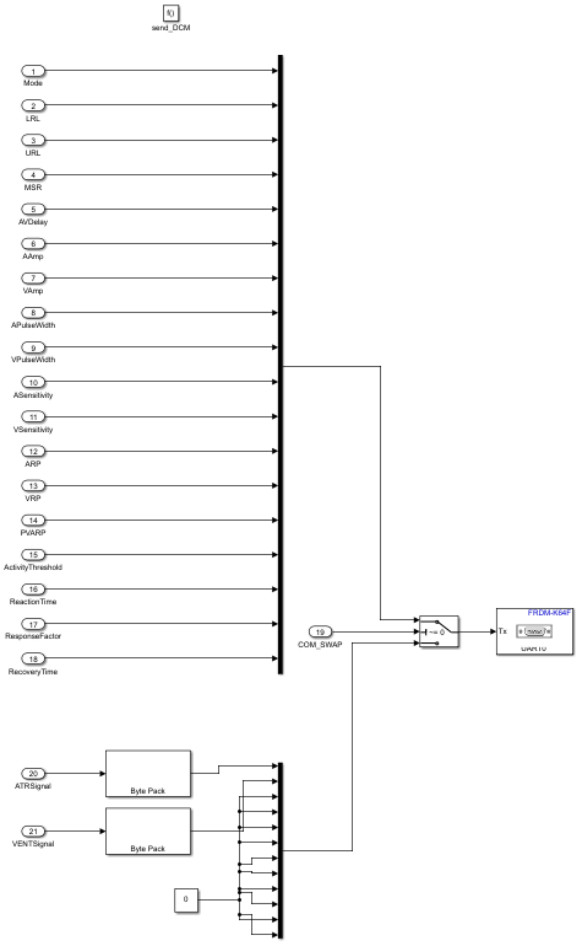


Figure 10

The setting of this switch is determined by the way send_DCM() is called. If it is called to send current variables, the switch will allow the top 18 variables to be packaged and sent through the TX port. If the function is called in egram mode, the switch will flip to allow the ATRsignal and VENTsignal to be packaged (along with zeros to fill the rest of the package) and sent using the TX port. Note: ATRsignal and VENTsignal are taken directly from pins on the pacemaker board.

Serial Verification

In order to verify that the byte pack is sent correctly, a function called send_Data_checked() was added. The function calls send_Pacemaker() and receive_Pacemaker() and compares the two values. If there are any differences, the user is notified with an error statement

```
def send_Data_checked( type, pacing_mode_value, saved_parameters):
    sent_values = send_Pacemaker(type, pacing_mode_value, saved_parameters)
    time.sleep(1)
    sent_values.pop(0)
    sent_values.pop(0)
    sent_values = sent_values[:4] + sent_values[5:]
    checker = recieve_Pacemaker()
    checker = checker[:4] + checker[5:]
    convert_to_int(sent_values)
    convert_to_int(checker)
    print("values", sent_values)
    print("check", checker)

    if (sent_values == checker): #need to change simulink serial
        print("sent packets verified")
    else:
        send_Pacemaker("default", pacing_mode_value, saved_parameters)
        print("Invalid bytes sent")
```

Rate Adaptive Pacing

According to requirements, all rate adaptive modes must track activity utilizing the on-board accelerometer and increase the pulse rate accordingly. In order to accomplish this, the data output from the accelerometer needs to be processed into a usable state.

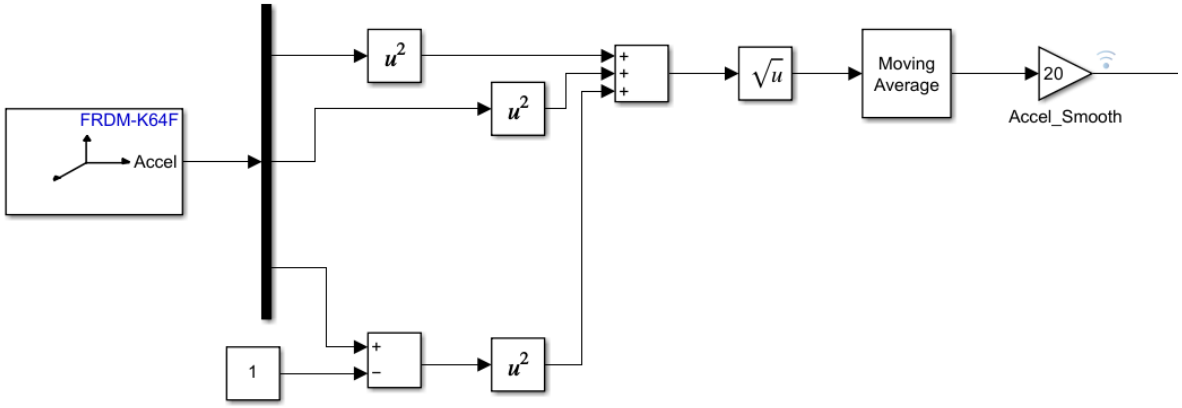
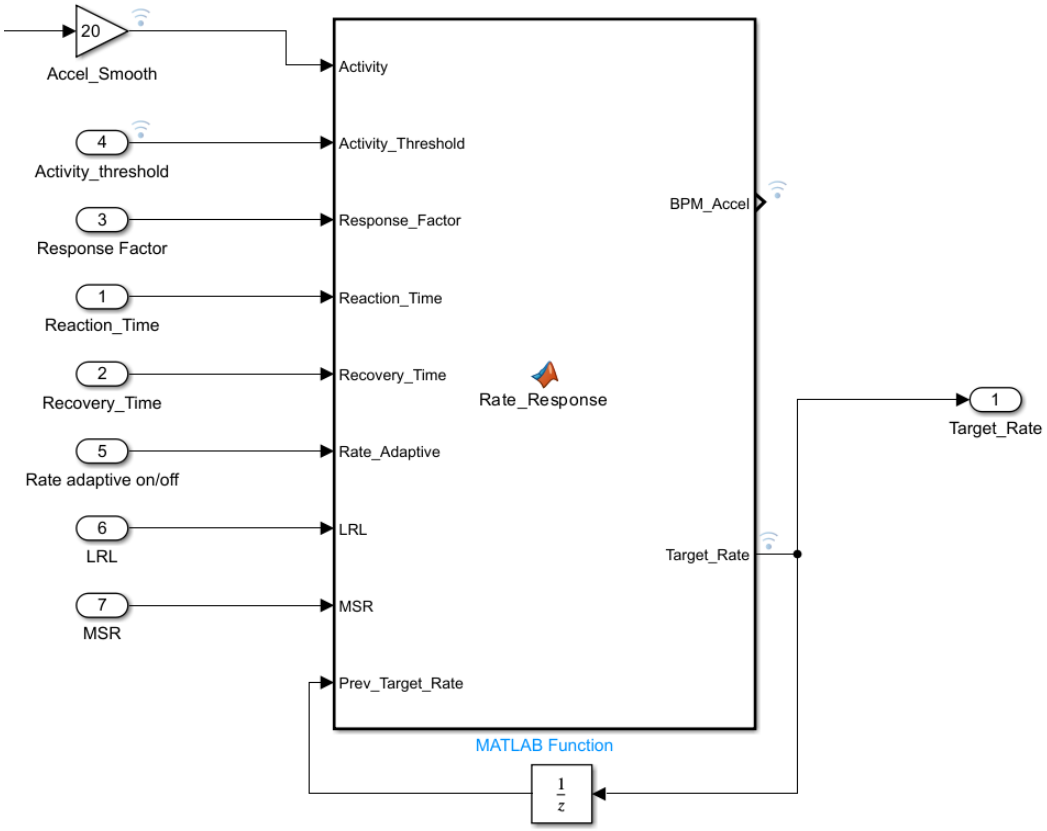


Figure 11. Accelerometer Data Signal Processing

Acceleration data is output as a 1 by 3 vector of double values for each axis of travel; x, y and z. Rather than being in m/s^2, these values are in g's (1g = 9.81 m/s^2), and due to the fact that there is a consistent 1g of acceleration in the z axis due to gravity, 1 is subtracted from the value to zero it out. The magnitude of the acceleration is then found by squaring the acceleration in each axis, adding them together, and square rooting them. As this magnitude is taken in real time, it shifts drastically as the accelerometer is rather sensitive. A moving average is then taken of the magnitude over a period of 300 samples, helping smooth the data. The averaged data then has a gain of 20 applied to it to scale the values into a usable range. Both the 300 samples of the moving average and the gain of 20 were found experimentally.




```
Activity_Factor = 10*Response_Factor*(Activity-Activity_Threshold);

derivative_Activity=diff(Activity);

if(Rate_Adaptive==0)
    BPM_Accel=0;
elseif(derivative_Activity>=0)              %if increasing
    BPM_Accel=Activity_Factor/Reaction_Time;
else                                        %if decreasing
    BPM_Accel=Activity_Factor/Recovery_Time;
end

Target_Rate=BPM_Accel*(1e-3)+Prev_Target_Rate;

if(Target_Rate<LRL)
    Target_Rate=LRL;
elseif(Target_Rate>MSR)
    Target_Rate=MSR;
elseif(Rate_Adaptive==0)
    Target_Rate=LRL;
end
```

Figure 12. Target Rate Calculations

Once the signal is processed, we then input the “Activity” level into a function alongside other constants sent from the DCM to calculate our target rate. First, the set Activity_Threshold is subtracted from Activity, then multiplied by a flat gain of 10 (found experimentally) and the Response_Factor. This value, Activity_Factor, will be positive when Activity is above Activity_Factor and negative otherwise. The derivative of Activity is also taken to figure out whether Activity is currently increasing or decreasing.

From here, an if statement is utilized to calculate BPM_Accel, or the acceleration of . If the mode is not rate adaptive, the acceleration is 0, and therefore the Target_Rate won't change from the LRL. If derivative_Activity is positive, then Activity_Factor is divided by Reaction_Time as our BPM will be increasing to the MSR. If derivative_Activity is negative, then Activity_Facrot is divided by Recovery_Time, as our BPM will be decreasing back to the LRL. The target rate is then calculated by multiplying the acceleration by the step-time of the program, converting it to the change in BPM that will occur during the current cycle. This change in BPM is then added to the Prev_Target_Rate, which is the Target_Rate from the previous cycle. If this new target rate drops below the LRL or above the MSR, it is corrected back within the bounds to prevent the Target_Rate from continuously increasing or decreasing. A failsafe is also implemented at the end of the program, where if the pacemaker isn’t currently in a rate adaptive mode, the Target_Rate is set to the LRL.

Because Target_Rate is the BPM value that the pacemaker should be pacing at, and rate adaptive functions can be turned on and off from within this subsystem, no new modes needed to be created specifically for the rate adaptive counterparts. Instead, Target_Rate replaced LRL within the modes. If rate adaptive is off, then the Target_Rate is set to the LRL, but when it is turned on, the Target_Rate unlocks and can change according to the accelerometer readings.

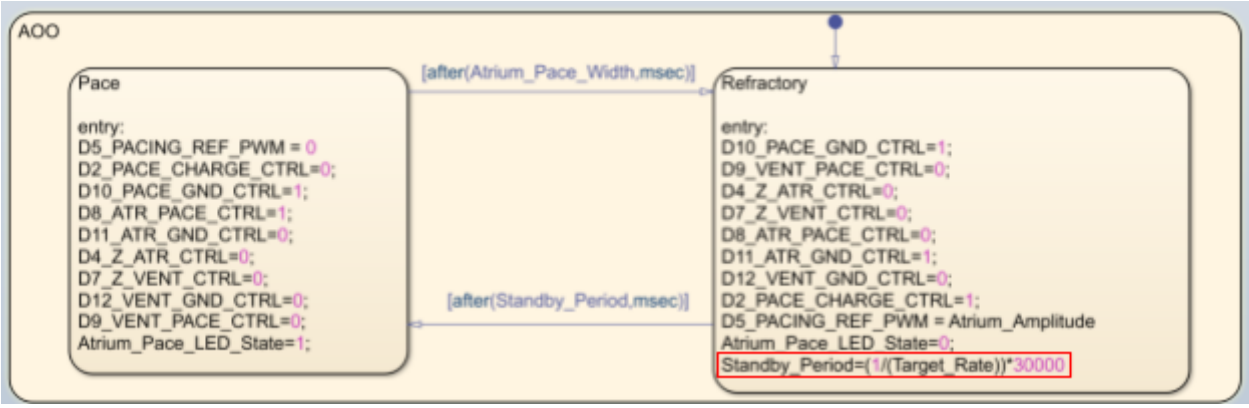


Figure 13. Implementation of Target_Rate within AOO

DCM Design

The development of the overall DCM design was separated into 5 distinct, interconnected categories that combined to create the DCM. Outside of input and output variables, each module is independent from each other and conducts their programs separately. Figure 14 shows our representation of our overall DCM design which involved 4 sections: login, registration, main, parameters.

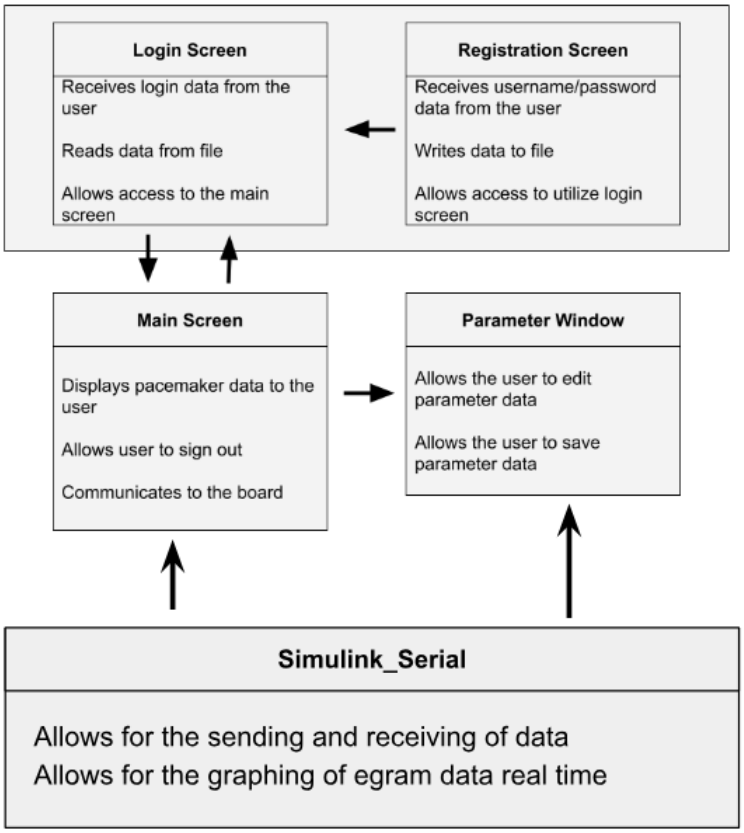


Figure 14. Overall DCM Design Structure

Black Box Model

After properly documenting the requirements and specifications, we used a black-box model to represent our DCM design solely in terms of the inputs and outputs. This simplification makes it easier to account for what parameters need to be extracted from the user and focus on which variables should be the end goal. Everything in-between including the error-handling, data storage, serial communication and simulink response is considered to be within the “black box” in this case.

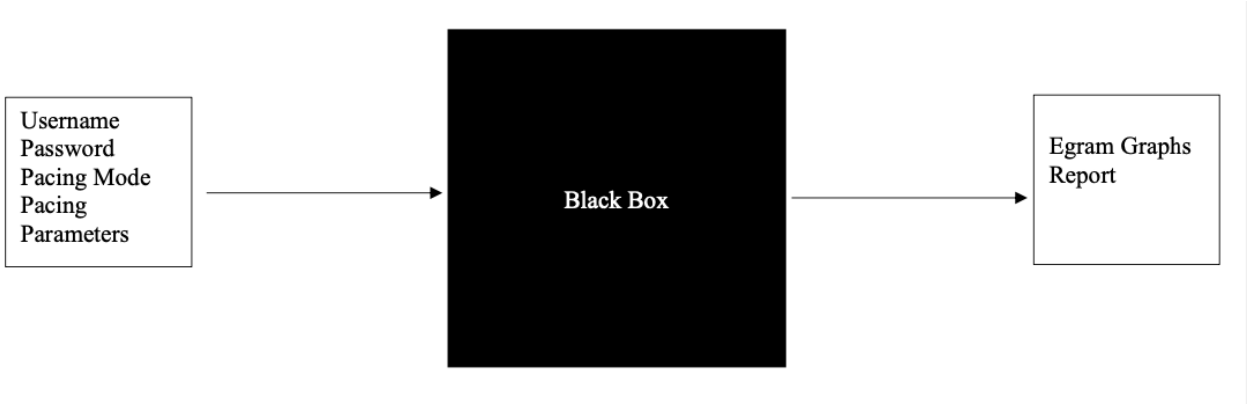


Figure 15. DCM Black Box Model Diagram

Figma Preliminary Design

Before we began programming the graphical user interface (GUI) of our DCM after making the requirements, we used Figma to model the different screens and visualize all of our requirements. This helped give us an idea of how to properly use the screen real estate and how many different screens we would need to create to optimize for user-accessibility.

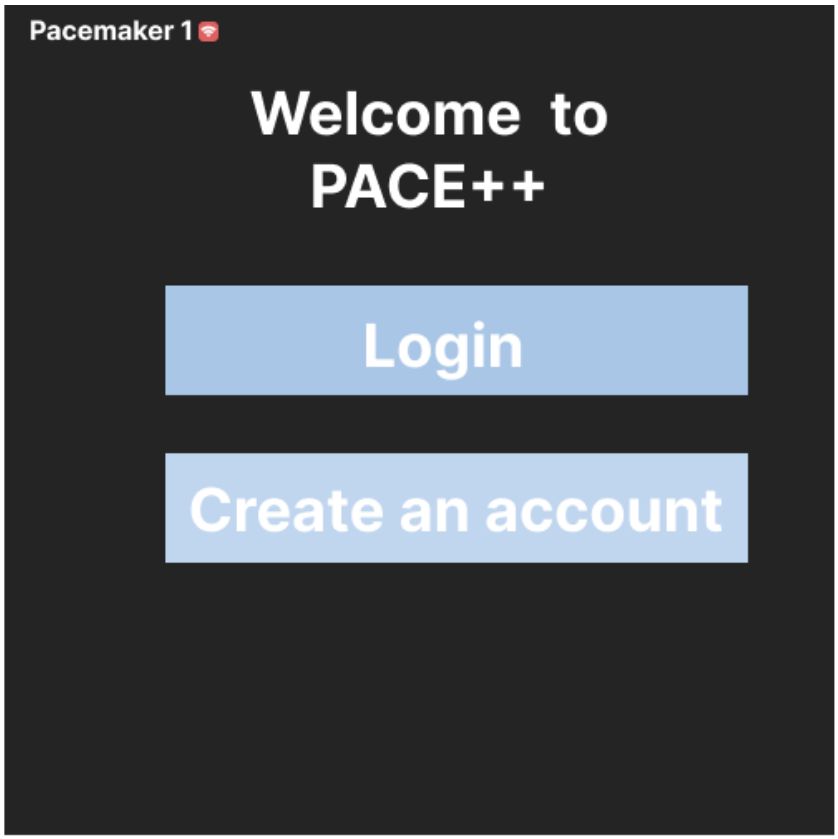


Figure 16. Figma Prototype Home Screen

Figure 17. Figma Prototype Parameters Screen

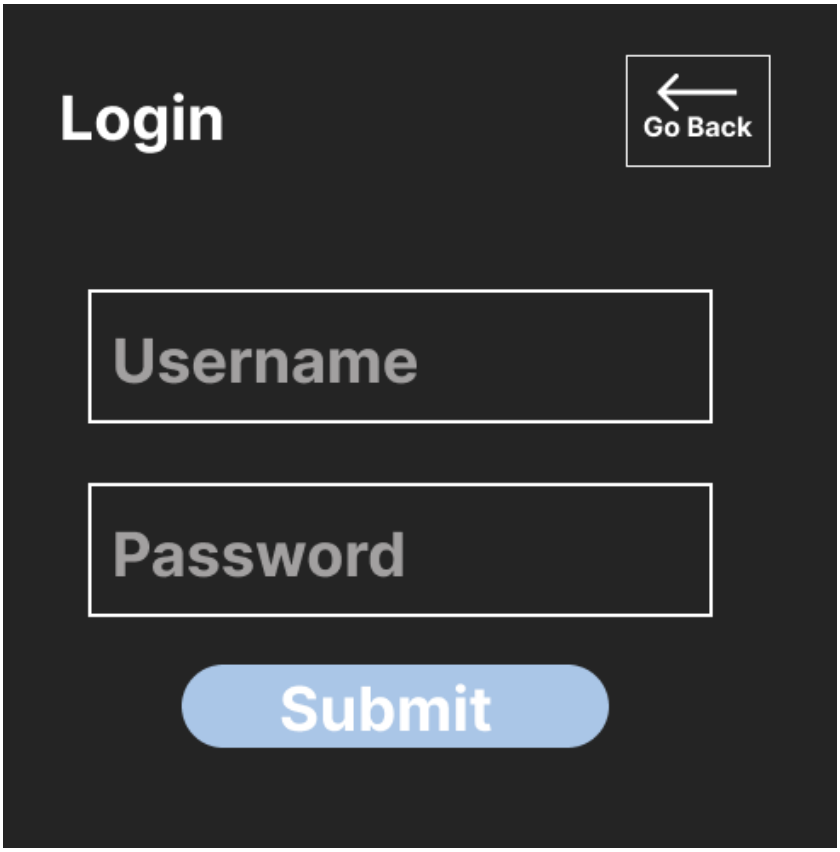


Figure 18. Figma Prototype Login Screen

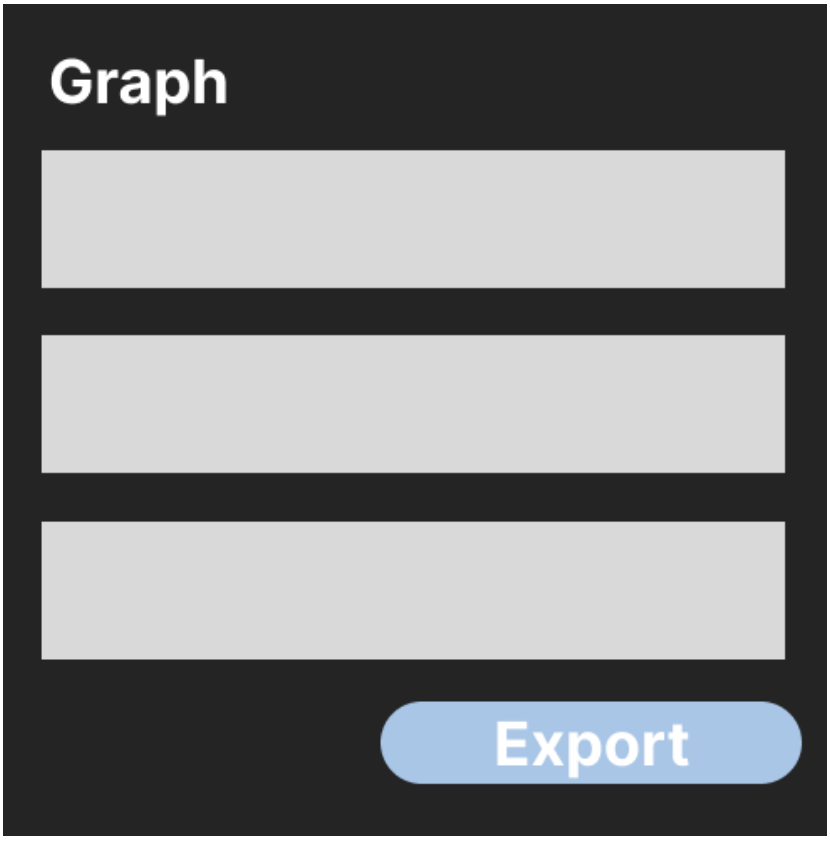


Figure 19. Figma Prototype Egram Graphs Screen

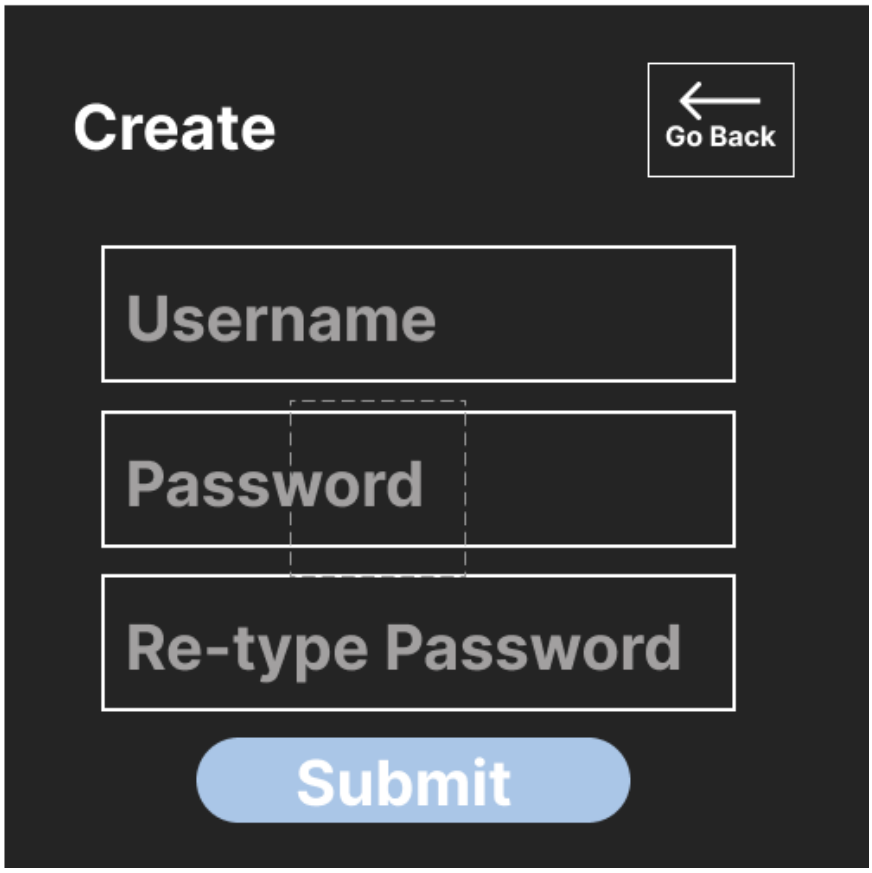


Figure 20. Figma Prototype Account Creation Screen

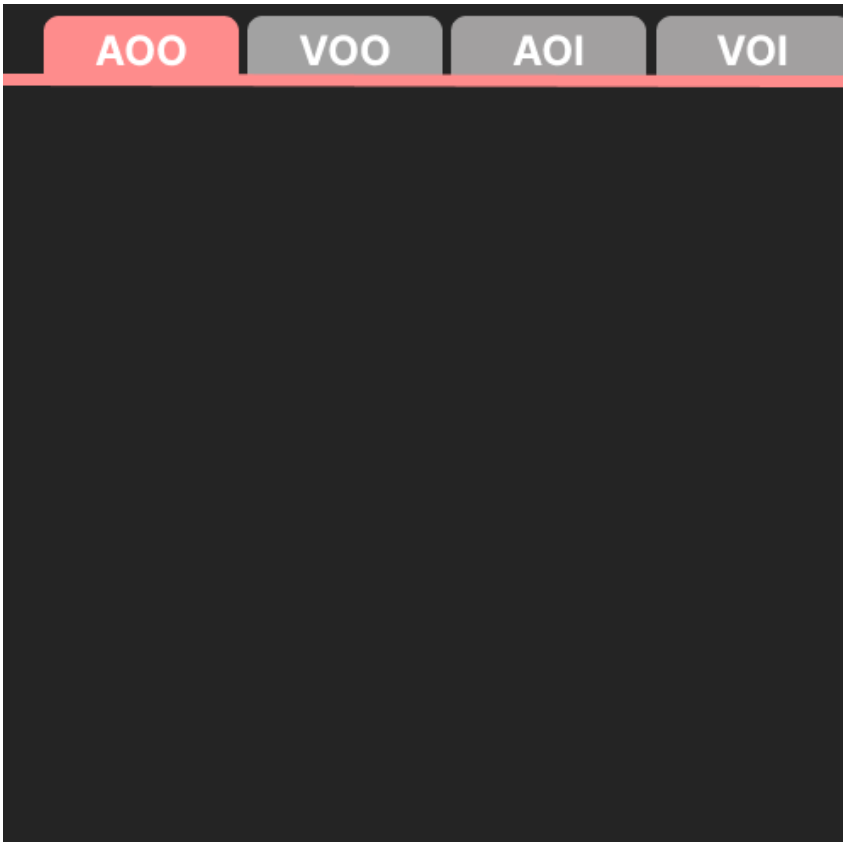


Figure 21. Figma Prototype AOO Mode Parameters Screen

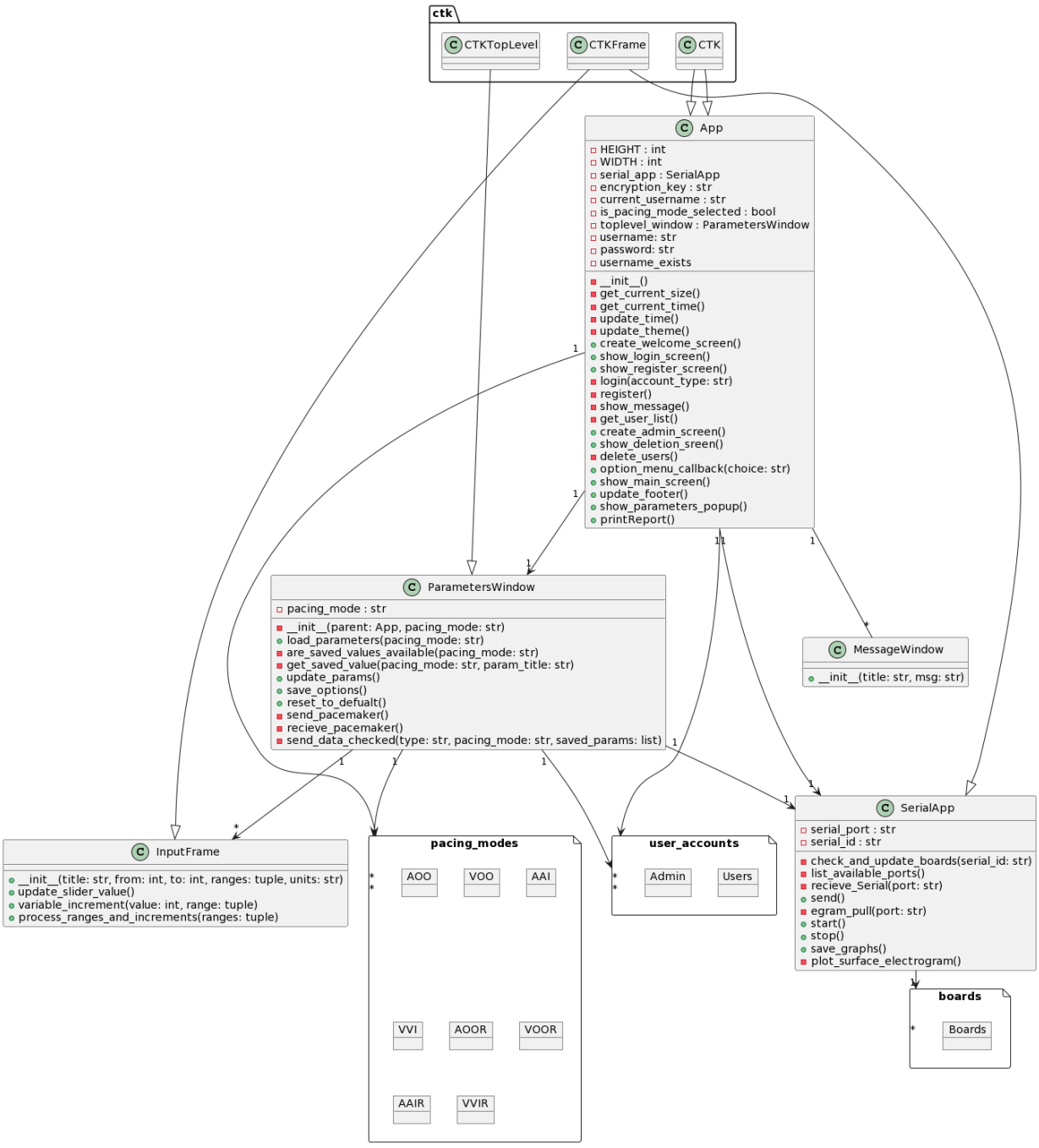
Figure 22. Figma Prototype VOO Mode Parameters Screen

Figure 23. Figma Prototype AOI Mode Parameters Screen

Figure 24. Figma Prototype VOI Mode Parameters Screen

Class Diagram

From our above diagrams, we implemented our design structure and black box into a class structure that we felt encapsulated our overall goals for what our design should function.



Repository

To streamline our DCM design, we decided to create a GitHub repository that allows us to easily make changes to our DCM. Our overall structure aimed to have high modularity and cohesion while maintaining low coupling. One main python file titled “app.py” included the code implementation for all the user-interface and user display features. Source and temp files were branched off around it to be imported into our final structure. Separate Python files were made for encryption, serial communication, To prevent data loss and git conflicts between different users we utilized separate GitHub branches to create certain features and merged them through pull requests to streamline our process.

Imported Libraries and Modules

Library Name	Description
os	Used for text file reading and writing to store the accounts and the saved parameters
customtkinter	Custom UI-based fork of python’s tkinter library to create the pacemaker GUI
tkinter	Standard tkinter library used for creating user interfaces and graphical elements in Python applications.
hashlib	Encryption to secure user data in the text file itself
PIL	A library for opening, manipulating, and saving many different image file formats.
datetime	A Python module for manipulating dates and times in both simple and complex ways. It provides classes for manipulating dates and times in both simple and complex ways.
time	A module that provides various time-related functions. It’s used to report the date.
base64	A module for encoding binary data into ASCII characters.
json	A lightweight data interchange format inspired by JavaScript object literal syntax. The json module allows for easy parsing of JSON data into Python data structures, and vice versa. It’s used to store our default and saved parameters.
platform	A module used to access underlying platform’s identifying data, such as, hardware, operating system, and interpreter version information. It’s used to determine the

	operating system used to prescribe the appropriate directory path.
cryptography	A package designed to expose cryptographic recipes and primitives. It's commonly used for secure data storage, secure communication, and authentication. The Fernet module was imported specifically so that its "generate_key()" method could be used to generate an encryption key for the account passwords.
matplotlib	Plotting library for creating static, interactive, and animated visualizations in Python. Specifically, the "FigureCanvasTkagg" function provided a way to display plots from the matplotlib library into tkinter.
struct	A module for converting between Python values and C structs represented as Python bytes objects. It's used for handling binary and ASCII data.
re	The regular expression module for Python. It provides Perl-style regular expression patterns, which are used for complex string matching and manipulation. It is used to use the "search" function to find the first occurrence of a string pattern in the available ports during serial communication.

Table 3: Imported libraries and modules

Classes Descriptions

Class Name	Description
InputFrame()	Each individual parameter label, slider, value display and unit display
MessageWindow()	Pop-up window for alerting the user about important alerts regarding the login and registration screens
ParametersWindow()	Pop-up window allowing the user to insert customer parameter values that differ from the default
App()	The main display of the pacemaker application, where the user can print the report, run the program, select their mode, and sign out of the program
SerialApp()	Class for handling serial communication between the pacemaker and DCM, specifically tailored for visualizing and managing electrogram data in real-time.

Functions

Each function is associated with the corresponding class name:

Function Name	Description
Input Frame()	
init_()	A frame containing a slider and value label for input parameters. Args: - master (Tkinter widget): The parent widget. - title (str): The title of the parameter. - from_ (float): The minimum value for the slider. - to (float): The maximum value for the slider. - ranges_and_increments (list or float): List of tuples specifying ranges and increments. - unit (str): The unit of the parameter.
update_slider_value()	Update the displayed value when the slider is moved. Args: - value (float): The current value of the slider.
variable_increment()	Adjust the value based on the specified ranges and increments. Args: - value (float): The current value. - rangeAndInc (list or float): List of tuples specifying ranges and increments. Returns: - float: The adjusted value.
process_ranges_and_imcrements( )	Process the input ranges to ensure it is a list of tuples. Args: - ranges (list or float): List of tuples specifying ranges and increments. Returns: - list: Processed ranges
MessageWindow()	
init_()	A simple message window. Args: - title (str): The title of the window. - msg (str): The message to display.
ParametersWindow()	
init_()	A window for setting and saving parameter values. Args:

	- app (App): The main application instance.
load_parameters()	Loads pacing parameters from a JSON file and creates input frames for each parameter. This method retrieves the selected pacing mode from a `StringVar` object and prints its value. It then locates and reads a JSON file ('pacing_modes.json') containing configuration data for various pacing modes. Based on the selected pacing mode, it extracts the corresponding parameter information and default values. The method clears any existing frames and checks if there are saved values available for the current pacing mode. For each parameter, it creates a new input frame with appropriate settings, including range and default values. If saved values are available and applicable, they are used; otherwise, default values from the JSON file are used. These frames are added to the 'frames' attribute of the class for display. Args: - pacing_mode (StringVar): A Tkinter StringVar object containing the selected pacing mode. Attributes modified: - frames (list): A list of frame widgets, each representing an input for a pacing parameter.
are_saved_parameters_available( )	Checks if saved parameter values are available for the given pacing mode. This method determines whether there are saved parameter values for a specific pacing mode associated with the current user. It first identifies the file path of 'user_accounts.json' located in the same directory as the script. The method then checks if a current user is set in the application. If a current user exists, the method opens and reads the 'user_accounts.json' file, which contains user data including saved pacing parameters. It iterates through the users in this file and, upon finding the current user, checks if there are saved parameters for the specified pacing mode. The method returns True if saved parameters are found, and False otherwise. Args: - pacing_mode_value (str): The value of the pacing mode to check for saved parameters. Returns: - bool: True if saved parameters are found for the given pacing mode and user, False otherwise.
get_saved_value()	Retrieves the saved value for a specific parameter and pacing mode from user data.

	This method looks up the saved value of a specified parameter for the given pacing mode, associated with the current user. It first determines the file path for 'user_accounts.json' and checks if there is a current user set. If a current user exists, the method opens and reads the 'user_accounts.json' file, which contains user profiles including their saved pacing parameters. It iterates through the user profiles to find the current user's profile. Upon locating the profile, the method checks if there are saved parameters for the specified pacing mode and, if so, retrieves the value for the given parameter. If the saved value is found, it is returned; otherwise, the method returns None. Args: - parameter_title (str): The title of the parameter for which the saved value is being retrieved. - pacing_mode_value (str): The pacing mode for which the parameter value is saved. Returns: - int or float or None: The saved value of the specified parameter if found, otherwise None.
update_parameters()	Updates the pacing mode parameters in the application. This method is responsible for updating the displayed pacing mode parameters. It calls the 'load_parameters' method, passing the provided 'pacing_mode' argument. The 'load_parameters' method handles the process of loading and displaying the relevant parameters for the selected pacing mode. This includes retrieving parameter values, either default or saved, and updating the display accordingly. Args: - pacing_mode (StringVar): A Tkinter StringVar object containing the selected pacing mode.
reset_to_default()	Reset parameter values to default and save to file.
send_Pacemaker()	Sends pacemaker configuration data to the pacemaker device based on the selected pacing mode. This method prepares and sends a list of pacemaker configuration parameters to the pacemaker device via a serial port. The configuration is dependent on the selected pacing mode and the operation type ('save'). It first defines a default list of configuration parameters and initializes the serial communication. Depending on the pacing mode and type, it modifies specific parameters in the list based on saved values. The modified list is then sent to the pacemaker device using the 'send' function. After sending the configuration, the method returns the updated list of parameters.

	Args: - type (str): The type of operation, e.g., 'save'. - pacing_mode_value (str): The selected pacing mode, e.g., 'AOO', 'AAI', etc. - saved_parameters (dict): A dictionary of saved parameter values for the given pacing mode. Returns: - list: The list of pacemaker configuration parameters that were sent.
convert_to_int()	Converts all elements in the provided array to integers. This method iterates through each element in the given array and converts each element to an integer. This is done in-place, meaning the original array is modified directly. The method does not return a value, as the conversion is applied to the array elements themselves. Args: - array (list): A list of elements (typically numeric or string representations of numbers) that will be converted to integers. Note: - The method expects the elements of the array to be convertible to integers. If any element cannot be converted, a ValueError will be raised.
recieve_Pacemaker()	Retrieves pacemaker configuration data from the pacemaker device. This method initializes a serial connection to the pacemaker device using the `SerialApp` class. It then defines a list of default values representing pacemaker configuration parameters. These values are sent to the pacemaker device using the `send` function with a specific function call value (34) indicating a request to retrieve data. The `send` function is expected to return the configuration data from the pacemaker. The method prints and then returns this received data array. Returns: - list: The list of configuration parameters received from the pacemaker.
send_Data_checked()	Sends pacemaker configuration data and verifies if the data was sent correctly. This method first sends configuration data to the pacemaker using the `send_Pacemaker` method. After a brief pause to ensure data transmission, it modifies the list of sent values to match the expected format for verification. The method then retrieves the current configuration from the pacemaker using `recieve_Pacemaker` and modifies this list similarly. Both lists are then converted to integers using `convert_to_int`. The method compares the sent values with the retrieved values to verify if the data was sent correctly. If the values match, a

	confirmation message is printed. If there is a discrepancy, it attempts to resend the data with a 'default' type and prints a message indicating an error in transmission. Args: - type (str): The type of operation, e.g., 'save'. - pacing_mode_value (str): The selected pacing mode, e.g., 'AOO', 'AAI', etc. - saved_parameters (dict): A dictionary of saved parameter values for the given pacing mode.
save_options()	Saves the current pacing mode parameters for the logged-in user and sends them to the pacemaker. This method first checks if a user is currently logged in. If a user is logged in, it retrieves the selected pacing mode and iterates through the frames containing parameter settings, collecting the titles and values of each parameter. These parameters are then saved in a dictionary. The method updates the user's saved parameters in 'user_accounts.json' with these new values. After updating the file, a success message is printed. It then calls 'send_Data_checked' to send these parameters to the pacemaker. If no user is logged in, a message is printed indicating that no user is logged in. Attributes modified: - saved_parameters (dict): A dictionary where keys are parameter titles and values are their corresponding settings, updated for the current pacing mode.
App()	
init()	Initializes the main application window with necessary configurations. This method sets up the main window for the application, specifying the title, size, and Screen dimensions. It initializes the `SerialApp` for serial communication and attempts to retrieve or generate an encryption key for secure data handling. The method also initializes various attributes such as a message window, the current username, a flag to track if a pacing mode is selected, and a toplevel window. It then calls ‘create_welcome_screen` to display the initial screen of the application. Additionally, it sets a minimum size for the application window. Attributes: - HEIGHT, WIDTH (int): The height and width of the screen. - serial_app (SerialApp): An instance of SerialApp for serial communication. - encryption_key (bytes): The encryption key used for secure data handling. - msg_window (widget): A widget for displaying messages, initialized as None.

	- current_username (str): The username of the currently logged-in user, initialized as None. - is_pacing_mode_selected (bool): A flag indicating if a pacing mode is selected, initialized as None. - toplevel_window (widget): A toplevel widget, initialized as None.
get_current_size()	Retrieves and returns the current screen width and height of the application window. This method updates the 'HEIGHT' and 'WIDTH' attributes of the class with the current screen height and width, respectively. It then prints these dimensions in a 'width x height' format. The method returns a tuple containing the current width and height. Returns: - tuple: A tuple containing two integers, where the first integer is the width and the second is the height of the application window.
get_current_time()	Retrieves and returns the current system time as a formatted string. This method obtains the current system time and formats it into a human-readable string representing the time in hours and minutes ('HH:MM' format). The formatted time string is then returned. Returns: - str: The current system time formatted as a string in 'HH:MM' format.
update_time()	Updates the provided label with the current time and schedules itself to run repeatedly. This method retrieves the current system time by calling `get_current_time`, which returns the time formatted as a string. It then updates the text of the given label widget with this current time. The method schedules itself to be called again after a delay of 1000 milliseconds (1 second), ensuring that the label is continuously updated with the current time every second. Args: - label (widget): The label widget to be updated with the current time. Note: - This method uses the 'after' method of the tkinter framework to schedule itself for repeated execution, ensuring that the time displayed is always up-to-date.
update_theme()	Checks the current appearance mode and returns a symbol representing the theme. This method determines the current appearance mode using the `ctk.get_appearance_mode()` function from a custom tkinter (ctk) library.

	It checks if the appearance mode is set to 'Dark'. If the mode is 'Dark', it returns a moon symbol (' ☾ ') representing the dark theme. Otherwise, it returns a sun symbol ('☀') representing a light theme. Returns: - str: A symbol (' ☾ ' for dark mode or '☀' for light mode) representing the current theme.
create_welcome_screen()	Create and display the welcome screen.
show_login_screen()	Switch to the login screen.
show_register_screen()	Switch to the register screen.
register()	Register a new user.
login()	Log in an existing user.
show_message()	Display a message window. Args: - title (str): The title of the message window. - msg (str): The message to be displayed.
get_user_list()	Retrieve the list of user accounts from the file. Returns: - list: List of user accounts.
username_exists()	Check if a username already exists. Args: - username (str): The username to check. Returns: - bool: True if the username exists, False otherwise.
back_to_welcome()	Return to the welcome screen.
create_admin_screen()	Creates the admin login screen for the application. This method sets up a dedicated frame for the admin login screen. It first hides the welcome frame and then initializes a new frame ('admin_frame') for the admin login interface. This frame includes a label indicating it is an admin login screen, entry fields for the admin username and password, and login and back buttons. The password entry field is configured to mask the entered password for security. Additionally, the method loads and sets background images for light and dark themes. The admin screen is designed to allow administrators to log in to access administrative functionalities. Components: - Admin login label. - Username entry: A field for entering the admin username. - Password entry: A masked field for entering the admin password. - Login button: Triggers the login process for the admin.

	- Back button: Allows returning to the welcome screen. - Background images: Themed images for the admin frame.
back_to_admin()	Returns to the admin login screen from the current interface. This method is responsible for navigating back to the admin login screen. It first clears all current widgets from the window by iterating through the window's children widgets and calling 'pack_forget' on each to remove them from the view. The method then resets the flag 'is_pacing_mode_selected' to None, indicating that no pacing mode is currently selected. Finally, it calls the 'create_admin_screen' method to set up and display the admin login screen. Note: - This method is typically used to provide a back navigation option from other interfaces within the application, allowing the user to return to the admin login screen.
deletion_checkbox_event()	Handles the event triggered by toggling a deletion checkbox associated with a user. This method is designed to respond to checkbox toggle events in a user management interface, particularly for selecting users for deletion. When a checkbox corresponding to a user is toggled, this method is invoked. It prints a message indicating the username associated with the toggled checkbox and the current state of the checkbox (checked or unchecked). The state of the checkbox is determined by the value of 'check_var', a variable typically linked to the checkbox's state. Args: - username (str): The username associated with the checkbox being toggled. - check_var (Tkinter Variable): A variable (e.g., IntVar, StringVar) linked to the checkbox, representing its state.
show_deletion_screen()	Displays the user deletion screen with options to select and delete users. This method sets up a dedicated frame for deleting users. It first hides the admin frame and then initializes a new frame ('deletion_frame') for the user deletion interface. The method retrieves a list of users and creates a checkbox for each user, allowing the admin to select users for deletion. Each checkbox is associated with a StringVar that tracks its state (checked or unchecked) and is linked to the 'deletion_checkbox_event' method to handle toggle events. The screen also includes 'Delete' and 'Back' buttons. The 'Delete' button is linked to the 'delete_users' method for initiating the deletion process, while the 'Back' button allows returning to the admin screen.

	Components: - A list of checkboxes, each representing a user. - 'Delete' button: Initiates the deletion of selected users. - 'Back' button: Navigates back to the admin screen.
delete_users()	Deletes the selected users from the system. This method identifies users selected for deletion based on the state of checkboxes in the deletion interface. It first determines the indices of users marked for deletion. If no users are selected, it prints a message and exits the method. Otherwise, it reads the existing user data from 'user_accounts.json', filters out the users marked for deletion, and writes the updated user list back to the file. The method also updates the user interface by removing the checkbox widgets and their corresponding StringVar objects for the deleted users. A confirmation message is printed once the deletion process is complete. Note: - The method assumes that each checkbox in 'checkboxList' corresponds to a user in the 'Users' list of the 'user_accounts.json' file, and their indices are aligned.
checkbox_event()	Handles the event triggered by toggling a checkbox. This method is designed to respond to the event of a checkbox being toggled in the user interface. It prints a message indicating that the checkbox has been toggled and displays the current state of the checkbox. The state of the checkbox (checked or unchecked) is determined by the value of 'self.check_var', which is a variable typically linked to the checkbox's state. Note: - 'self.check_var' should be a Tkinter variable (e.g., IntVar, StringVar) that is associated with the checkbox's state.
optionmenu_callback()	Callback function for the option menu. Args: - choice (str): The selected option.
pacing_mode_callback()	Responds to changes in the pacing mode selection and updates the interface accordingly. This method is called when there is a change in the pacing mode selection. It retrieves the currently selected pacing mode and sets the flag 'is_pacing_mode_selected' based on whether a valid pacing mode (other than "Select Pacing Mode") is selected. If a top level window exists and is open, it is closed. If a valid pacing mode is selected, the method creates and displays

	a new 'ParametersWindow' to allow the user to configure settings for the selected pacing mode. Args: - *args: Variable length argument list, used to handle any number of arguments passed to the callback. Note: - 'self.pacing_mode' should be a Tkinter variable (e.g., StringVar) associated with the pacing mode selection. - 'self.toplevel_window' is used to create a new window for pacing mode parameters.
show_main_screen()	Switch to the main screen.
update_footer()	Updates the footer information of the application interface and schedules regular updates. This method updates the text of the 'board_label' widget with the current version of the application, the serial ID of the connected device, the board number to which the device is connected (determined by calling 'check_and_update_boards'), and a reference to McMaster University. The update is performed at regular intervals (every 105 milliseconds) by scheduling the 'update_footer' method to be called again using the 'after' method, ensuring the footer information remains current. Note: - 'self.board_label' is assumed to be a label widget in the application's interface. - 'self.serial_app.serial_id' provides the current serial ID of the connected device.
theme_event()	Toggle between dark and light themes.
show_parameters_popup()	Display the parameters window.
verify_connection()	Verifies the serial connection status and updates the connection label in the application. This method checks the status of the serial connection by referencing 'self.serial_app.serial_port'. It then updates the 'connection' label with the current connection status, including the port number to which the application is connected. The text color of the label is set to a specific shade of green (hex color code '#1cac78') to visually indicate a successful connection. The method is scheduled to run repeatedly every 500 milliseconds using the 'after' method, ensuring that the connection status is continuously monitored and updated in the application interface. Note: - 'self.connection' is assumed to be a label widget in the application's interface that displays the connection status.

	- The method continuously monitors the connection and updates the interface accordingly.
printReport()	Generates and saves a detailed HTML report for the pacemaker data. This method compiles a comprehensive report in HTML format, detailing various parameters and graphical representations associated with the pacemaker data. It includes the date and time of report generation, serial ID, and pacing mode information, along with retrieved pacemaker parameters and graphical data like egram graphs. The HTML report contains styled tables, images of graphs, and descriptive text. The method saves the generated report as an HTML file ('PACE++_Printed_Report.html'). A confirmation message is printed once the report is successfully generated. Components of the Report: - Header information including the institution name, date/time, version, and serial number. - Bradycardia parameters table with values and units. - Graphical representations of atrial and ventricle outputs, combined egram data, and surface electrogram. - Additional sections as required. Note: - The method assumes that the necessary graphs (e.g., atrial, ventricle, combined, surface electrogram) are saved as PNG files and referenced in the HTML content.
generate_encryption_key()	Generates and returns a new encryption key. This method uses the Fernet module from the cryptography library to generate a secure symmetric encryption key. The generated key is suitable for use with Fernet symmetric encryption and decryption. The key is returned as a byte string. Returns: - bytes: A newly generated encryption key in byte string format. Note: - This method is typically used for generating keys for secure data encryption and should be stored securely.
encrypt_data()	Encrypts the provided data using the specified encryption key. This method takes a dictionary of data and an encryption key, and returns a new dictionary with the data encrypted. It utilizes the Fernet symmetric encryption provided by the cryptography library. Each value in the input dictionary is encrypted, and the encrypted values are stored in a new dictionary with the same keys. If a value is a list, it is first converted to a

	JSON string before encryption. The method returns this new dictionary containing the encrypted values. Args: - data (dict): A dictionary where each key-value pair represents the data to be encrypted. - key (bytes): The encryption key used for encrypting the data. Returns: - dict: A dictionary with the same keys as the input, but with encrypted values. Note: - This method is designed to handle string and list types. Lists are converted to JSON strings before encryption.
decrypt_data()	Decrypts the provided encrypted data using the specified encryption key. This method takes a dictionary of encrypted data and an encryption key, and returns a new dictionary with the data decrypted. It uses the Fernet symmetric decryption provided by the cryptography library. Each value in the input dictionary is decrypted, and the decrypted values are stored in a new dictionary with the same keys. If the key is 'roles', the decrypted value is parsed as JSON. The method handles exceptions by printing a message in case of an InvalidToken error, indicating possible data tampering. The method returns the new dictionary containing the decrypted values. Args: - encrypted_data (dict): A dictionary where each key-value pair represents the encrypted data to be decrypted. - key (bytes): The decryption key used for decrypting the data. Returns: - dict: A dictionary with the same keys as the input, but with decrypted values. Note: - The method specifically handles the 'roles' key by converting its decrypted value from JSON to a Python object. - The method catches and handles the InvalidToken exception, which may indicate tampering with the encrypted data.
save_encrypted_data_to_file()	Saves the provided encrypted data to a file. This method writes the encrypted data to a specified file. The encrypted data is passed in as a dictionary, and the method iterates over each key-value pair in the dictionary. For each pair, it writes the key and

	encrypted value, formatted as 'key: value', to the file. The data is written in binary mode to ensure the encrypted byte strings are correctly saved. Each key-value pair is written on a new line in the file. Args: - encrypted_data (dict): A dictionary where each key-value pair represents the encrypted data to be saved. - filename (str): The name of the file where the encrypted data will be saved. Note: - The method writes data in binary mode ('wb'), which is necessary for correctly saving encrypted byte strings.
read_encrypted_data_from_file()	Reads encrypted data from a file and returns it as a dictionary. This method opens a specified file in binary read mode and reads its contents line by line. Each line is expected to be formatted as 'key: value', representing a key-value pair of encrypted data. The method decodes each line, splits it into key and value components, and reconstructs the dictionary with these key-value pairs. The values are re-encoded to bytes before being stored in the dictionary. The method returns this dictionary containing the encrypted data. Args: - filename (str): The name of the file from which the encrypted data will be read. Returns: - dict: A dictionary containing the encrypted data, where each key-value pair represents a piece of encrypted data read from the file. Note: - The method reads the file in binary mode ('rb') and expects each line to be properly formatted with a key and value separated by ': '.
simulink_serial.py	
check_and_update_boards()	Check the existing board entries in a JSON file and update it with a new board if the given serial ID does not already exist. If the file does not exist, it creates a new one with the given entry. The function first attempts to open and read 'boards.json'. If the file is not found, it initializes an empty dictionary. It then checks if the given serial_id is already in the data. If it exists, the function returns the corresponding board number. If it doesn't exist, the function calculates the next available board number, adds the new serial_id with its corresponding board number to the dictionary, saves the updated dictionary back to 'boards.json', and returns the new board name.

	Args: - serial_id (str): The serial ID of the board to check or add. Returns: - str: The board number corresponding to the given serial ID, either retrieved or newly assigned.
list_available_ports()	List available COM ports and return the first port with a matching serial number in its hardware ID. Uses`serial.tools.list_ports.comports()` to get a list of available COM ports. It checks if there are any COM ports available. If not, it prints a message stating no COM ports are found. Otherwise, it iterates through the sorted list of ports. For each port, it searches for a serial number in the hardware ID using a regular expression. If a match is found, the function returns the port and the serial number. If no matching serial number is found in any of the COM ports, the function returns `None, None`. Returns: - tuple: A tuple of two elements where the first element is the port and the second element is the serial number if a matching serial number is found; otherwise, `None, None`.
recieveSerial()	Reads and unpacks serial data from a specified COM port, returning an array of parameters. The function initializes several variables with default values corresponding to different pacemaker parameters. It then attempts to open a serial connection on the provided port with a specified baud rate. It reads the incoming data based on a predefined struct format and unpacks this data into various pacemaker parameters. The parameters include Mode, Lower Rate Limit (LRL), Upper Rate Limit (URL), Maximum Sensor Rate (MSR), Atrioventricular (AV) Delay, Amplitudes (AAmp, VAmp), Pulse Widths (APulseWidth, VPulseWidth), Sensitivities (ASensitivity, VSensitivity), Refractory Periods (ARP, VRP), and additional parameters like Activity Threshold, Reaction Time, Response Factor, and Recovery Time. If a ValueError occurs during unpacking, it prints an error message. The function closes the serial connection before returning the data. Args: - port (str): The COM port to open for serial communication. Returns: - list: A list containing unpacked data representing various pacemaker parameters.
send()	Sends serialized data over a serial connection to a specified port and optionally receives a response. This function takes multiple pacemaker parameters, serializes them using a predefined struct format, and sends the

	serialized data over a serial connection to the given port. It converts the parameters to integers, adjusting their scales where necessary. The function then packs these parameters into a byte stream and writes them to the specified COM port. After sending the data, it closes the serial connection. If the `Function_call` parameter is 34, it calls 'receiveSerial(port)' to read a response from the serial port. Args: - Sync (int): Synchronization byte. - Function_call (int): Specifies the function to be called. - Mode, LRL, URL, MSR, AVDelay, AAmp, VAmp, APulseWidth, VPulseWidth, ASensitivity, VSensitivity, ARP, VRP, PVARP, ActivityThreshold, ReactionTime, ResponseFactor, RecoveryTime (int): Pacemaker parameters. - port (str): The COM port to open for serial communication. Returns: - list or None: Returns the response from `receiveSerial(port)` if `Function_call` is 34; otherwise, returns None.
egramPull	Sends a request over a serial connection to pull electrogram (egram) data, then receives and unpacks this data. The function first creates a struct for sending data with a predefined format, packs a specific message into this struct, and sends it over a serial connection to the given port. This message is intended to request egram data. After sending the request, it initializes another struct with a different format to receive and unpack the egram data. It reads the incoming data from the serial port, unpacks it, and prints certain elements of the unpacked data. Finally, the function closes the serial connection and returns the unpacked data. Args: - port (str): The COM port to open for serial communication. Returns: - tuple: A tuple containing the unpacked egram data.
SerialApp()	
init()	Initializes an instance of the class, setting up the serial port, graphs, and other GUI components. This method initializes the class by first calling the constructor of the superclass with any additional keyword arguments. It then sets up a serial port and ID by calling `list_available_ports()`. The method sets up three subplots using matplotlib: one for atrial output, one for ventricle output, and one for combined egram data. Each subplot is configured with titles, labels, and a legend. The plots are then packed into a tkinter canvas

	widget. Additionally, the method initializes data lists for storing graph data and a time counter for plotting, as well as setting a maximum length for these data arrays. Args: - master (widget): The parent widget. Default is None. - **kwargs: Arbitrary keyword arguments passed to the superclass constructor. Attributes: - serial_port, serial_id: Set by the `list_available_ports()` function. - fig, ax1, ax2, ax3: Matplotlib figure and axes for the graphs. - line1, line2, line1_combined, line2_combined: Line objects for the plot data. - canvas, canvas_widget: Tkinter canvas and widget for displaying the plots. - xdata, ydata1, ydata2: Lists to store the x and y data for the plots. - start_time: Time counter for the x-axis of the plots. - running: Boolean flag to indicate if the plotting is active. - max_length: Maximum length of the data arrays.
start()	Begins the graphing process if it is not already active. When invoked, this method first prints a message indicating that the graphing process has started. It checks if the 'running' attribute is False, implying that the graphing process is not currently active. If so, the method sets this attribute to True, marking the beginning of the graphing process. It records the current time as the start time for the graphing. The method then clears any existing data in the 'xdata', 'ydata1', and 'ydata2' lists, which are used to store plot data. Finally, it calls the 'update_plot' method to initiate the process of updating the plots with new data. Attributes modified: - running (bool): Set to True to indicate the graphing process is now active. - start_time (float): Records the current time as the start time of the graphing. - xdata, ydata1, ydata2 (list): Lists for storing plot data, cleared at the start of graphing.
stop()	Stops the graphing process. This method is used to stop the graphing process. When called, it prints a message indicating that the graphing process has been stopped. It then sets the 'running' attribute to False, which is used to control the active state of the graphing process. Attributes modified:

	- running (bool): Set to False to indicate the graphing process is no longer active.
update_plot()	Updates the plots with new data if the graphing process is active. This method is responsible for updating the atrial and ventricle graphs, as well as a combined egram graph, with new data. It first checks if the graphing process is active, indicated by the 'running' attribute. If it is active, the method calculates the current time relative to the start time and retrieves new egram data by calling `egramPull(self.serial_port)`. This data is appended to the respective data lists ('xdata', 'ydata1', and 'ydata2'). The method then updates the line objects for each plot with the new data, adjusts the axes limits, and autoscales the view. It also ensures that only the most recent data points up to a maximum number defined by 'max_length' are kept in the data lists. The canvas is redrawn to reflect the updated plots. Finally, the method schedules itself to be called again after a short delay, creating a continuous update loop. Attributes modified: - xdata, ydata1, ydata2 (list): Lists storing the data for the plots, updated with new egram data. - line1, line2, line1_combined, line2_combined: Line objects updated with new plot data.
update_available_ports()	Updates the list of available serial ports and schedules a periodic update. This method refreshes the list of available serial ports by calling `list_available_ports()`, which returns the first available port and its corresponding serial ID. These values are then assigned to the instance attributes 'serial_port' and 'serial_id'. After updating these attributes, the method schedules itself to be called again after a short delay (105 milliseconds), creating a continuous loop to periodically check for changes in the available serial ports. Attributes modified: - serial_port (str): The first available serial port. - serial_id (str): The serial ID corresponding to the serial_port.
save_graphs()	Saves the current atrial, ventricle, and combined egram graphs as PNG files. This method first ensures that the data displayed on the graphs is current by calling 'update_plot'. It then creates and configures separate matplotlib figures for the atrial graph, the ventricle graph, and the combined egram graph, each with titles, labels, and legends. The data from 'xdata', 'ydata1', and 'ydata2' are used to plot these graphs. After setting up each graph, the method saves them as PNG files ('Atrial_Graph.png', 'Ventricle_Graph.png', and 'Combined_Graph.png'). It closes each figure

	after saving to free up resources. A message is printed to indicate that the graphs have been saved. Saved Files: - Atrial_Graph.png: The saved atrial output graph. - Ventricle_Graph.png: The saved ventricle output graph. - Combined_Graph.png: The saved combined egram graph.
plot_surface_eleectrogram()	Creates and saves a surface electrogram graph based on the summed atrial and ventricle data. This method generates a surface electrogram by summing the atrial and ventricle data stored in 'ydata1' and 'ydata2', respectively. It creates a new matplotlib figure and plots this summed data as a surface electrogram. The plot is configured with a title, labels, and a legend. After setting up the graph, the method saves it as a PNG file ('Surface_Electrogram.png') and then closes the figure to free up resources. Saved File: - Surface_Electrogram.png: The saved surface electrogram graph.

DCM Version 1 GUI Screens

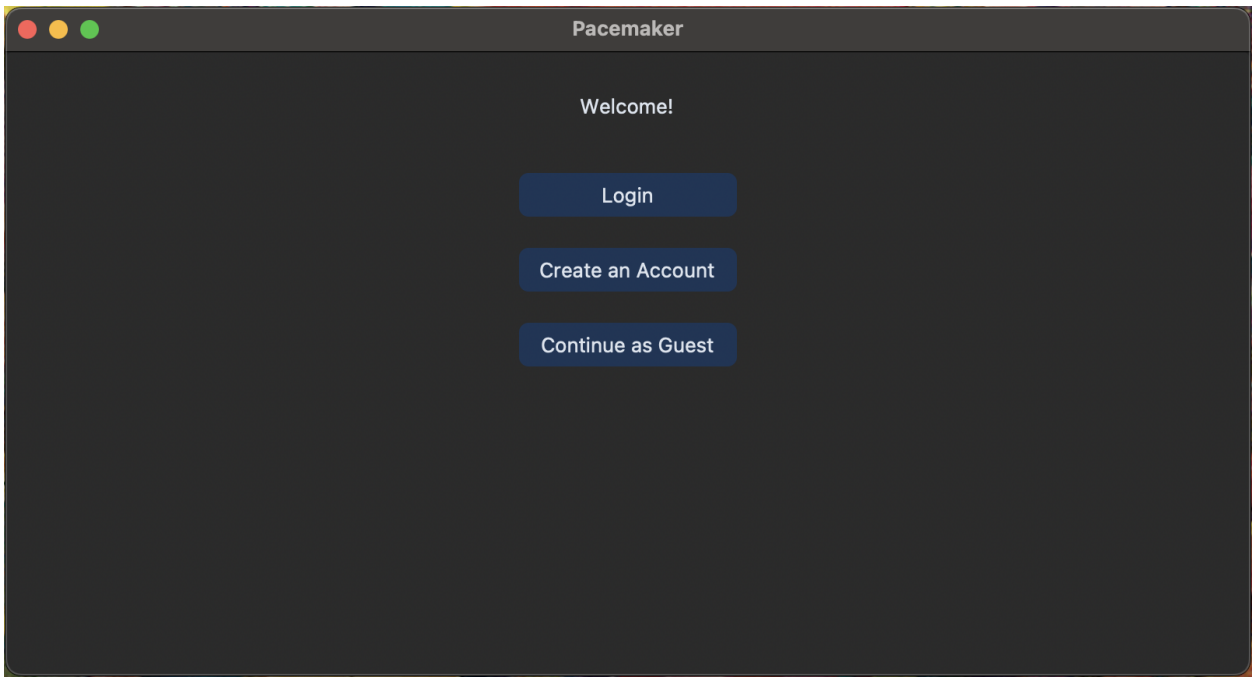


Figure 25. DCM - Version 1 Home Screen

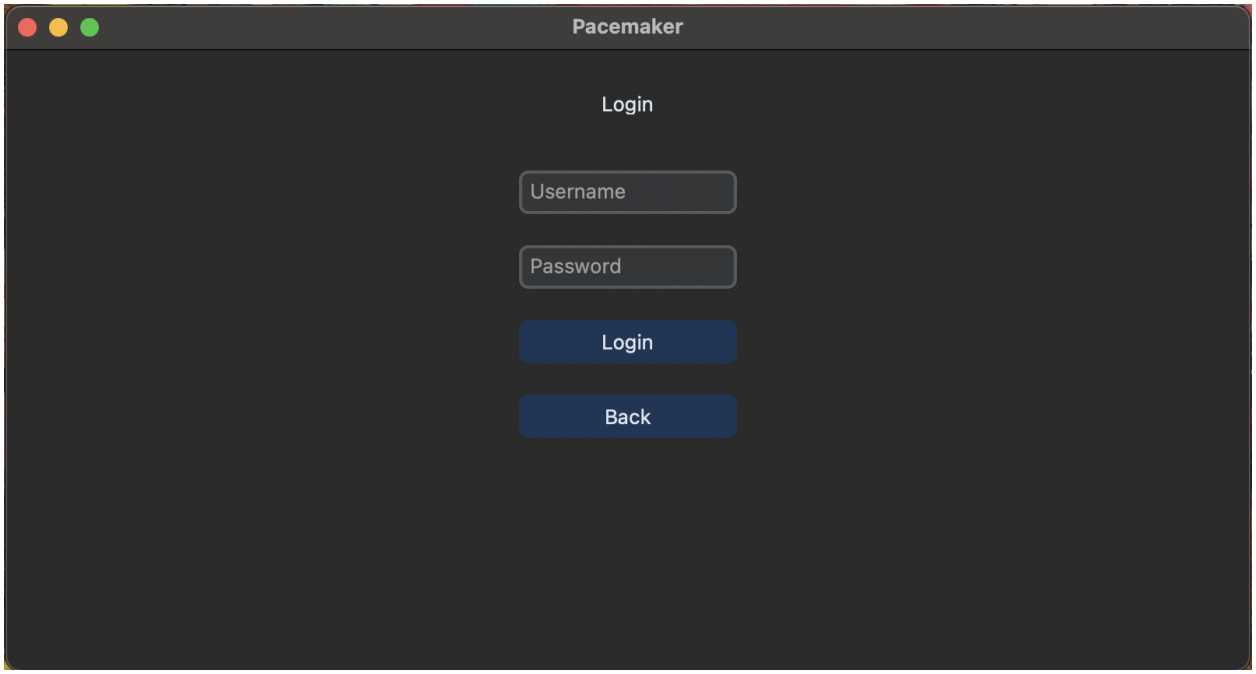


Figure 26. DCM - Version 1 Login Screen

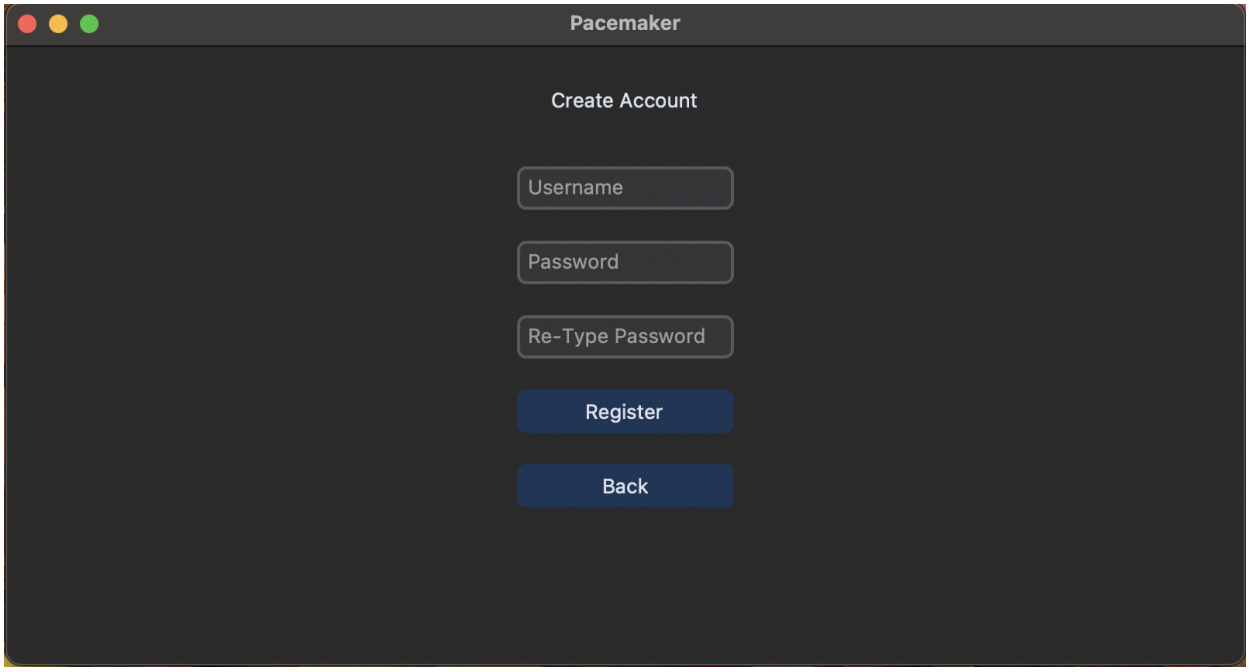


Figure 27. DCM - Version 1 Account Creation Screen

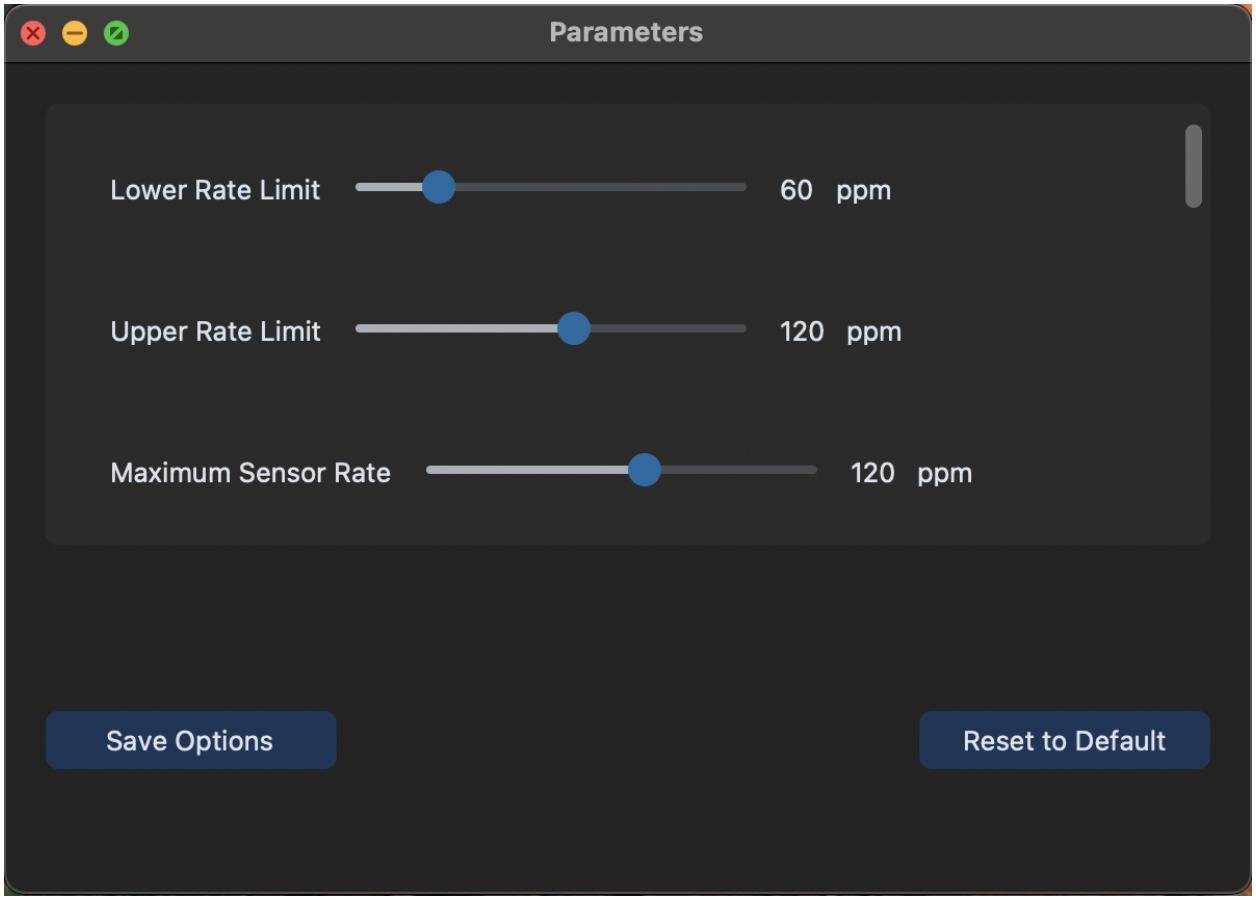


Figure 28. DCM Version 1 Parameters Screen

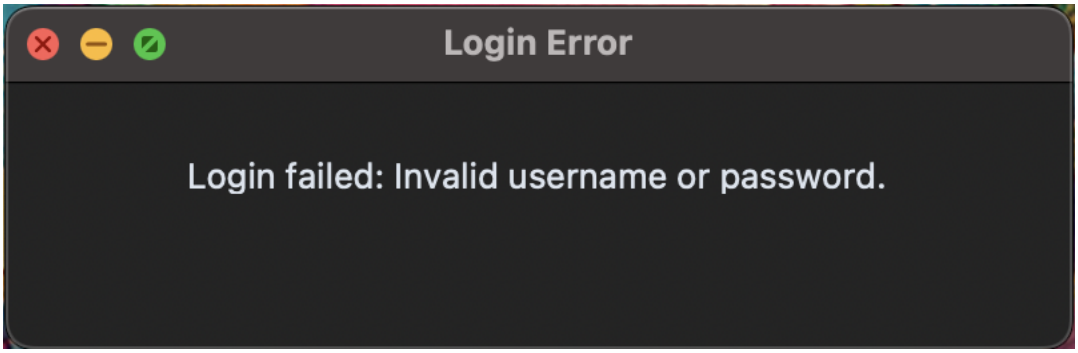


Figure 29. DCM - Version 1 Login Error Message Window

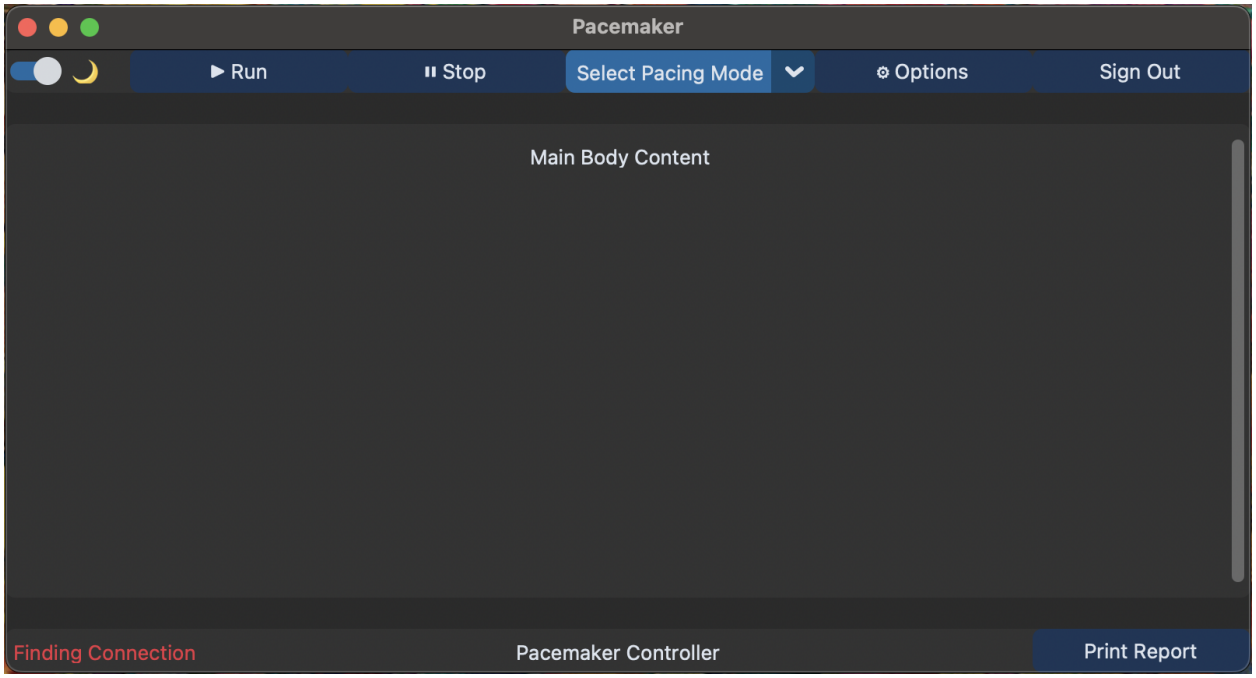


Figure 30. DCM - Version 1 Main App Interface Screen

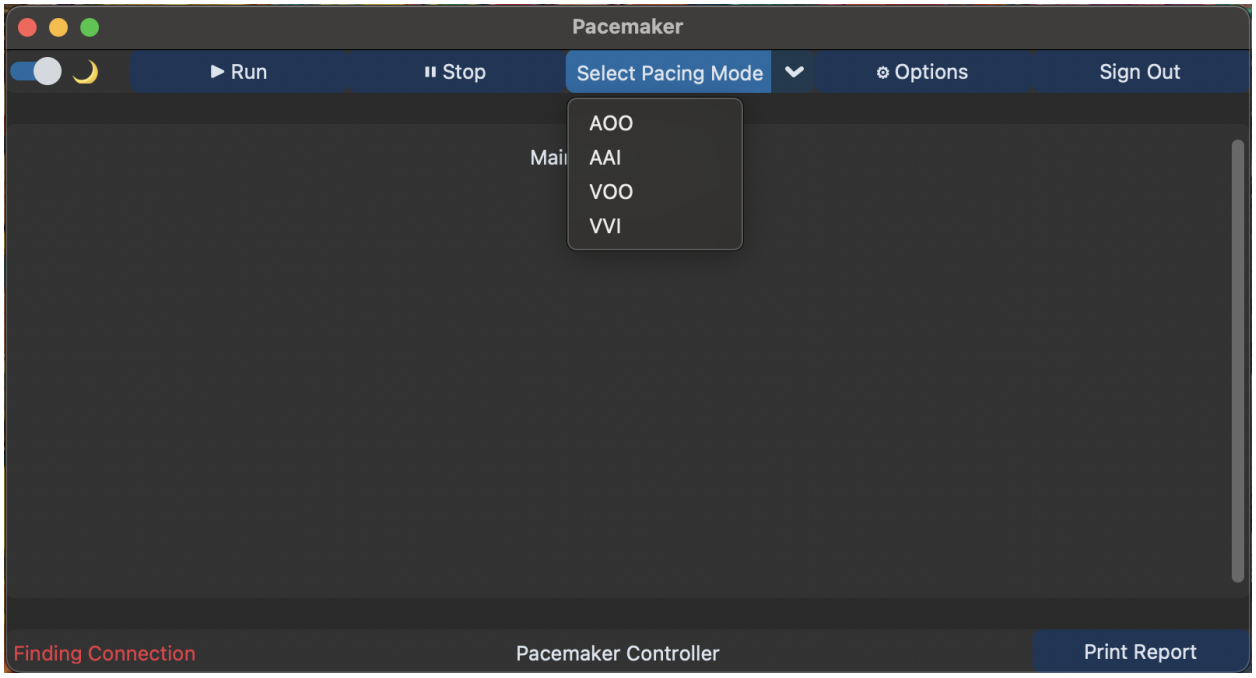


Figure 31. DCM - Version 1 Mode Selection Screen

DCM - Finalized GUI Screens

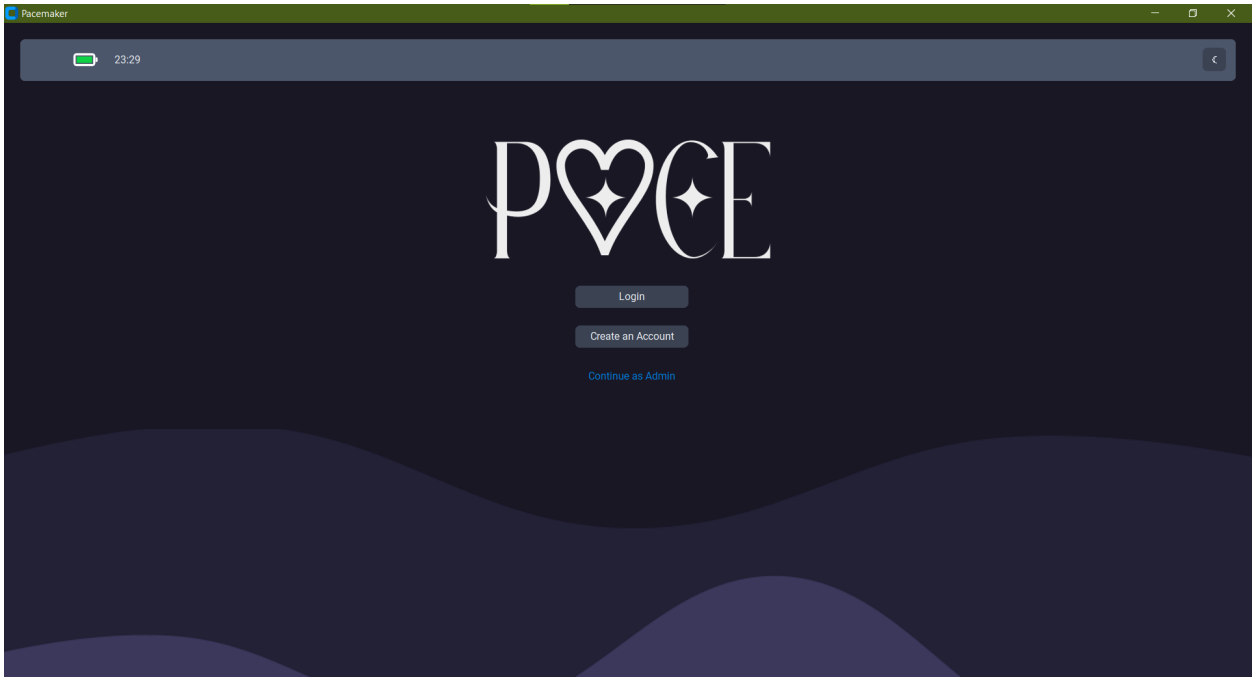


Figure 32. DCM - Finalized Home Screen (Dark Mode)

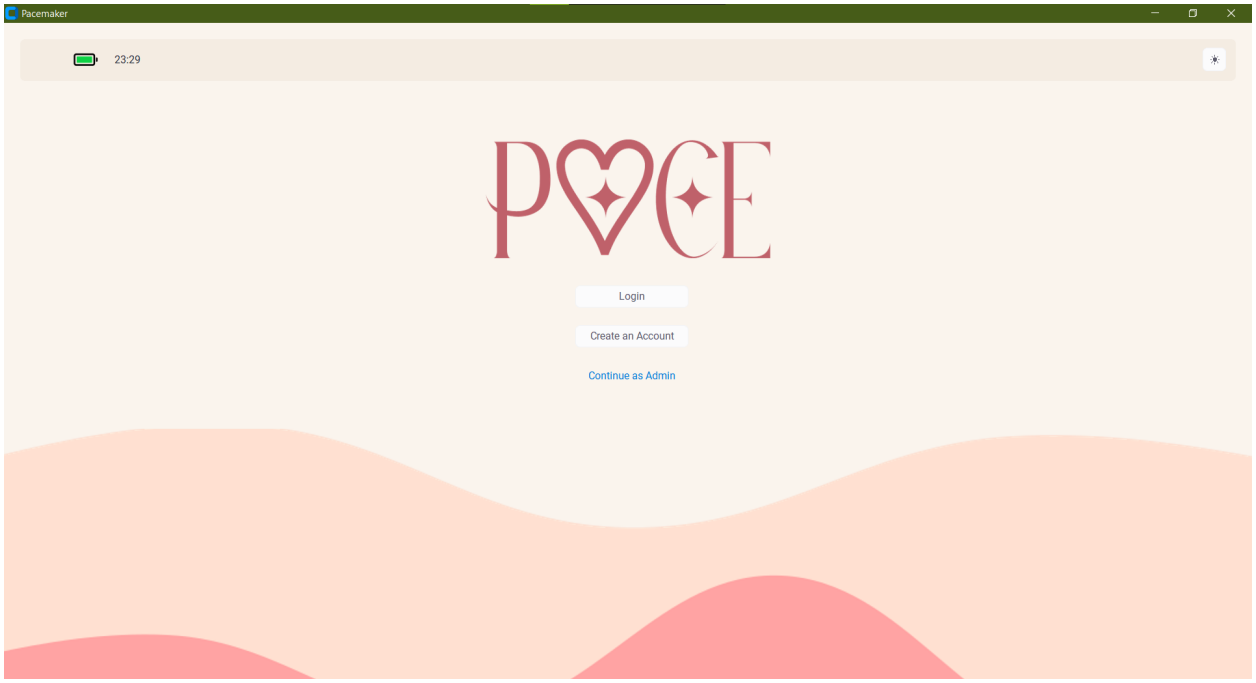


Figure 33. DCM - Finalized Home Screen (Light Mode)

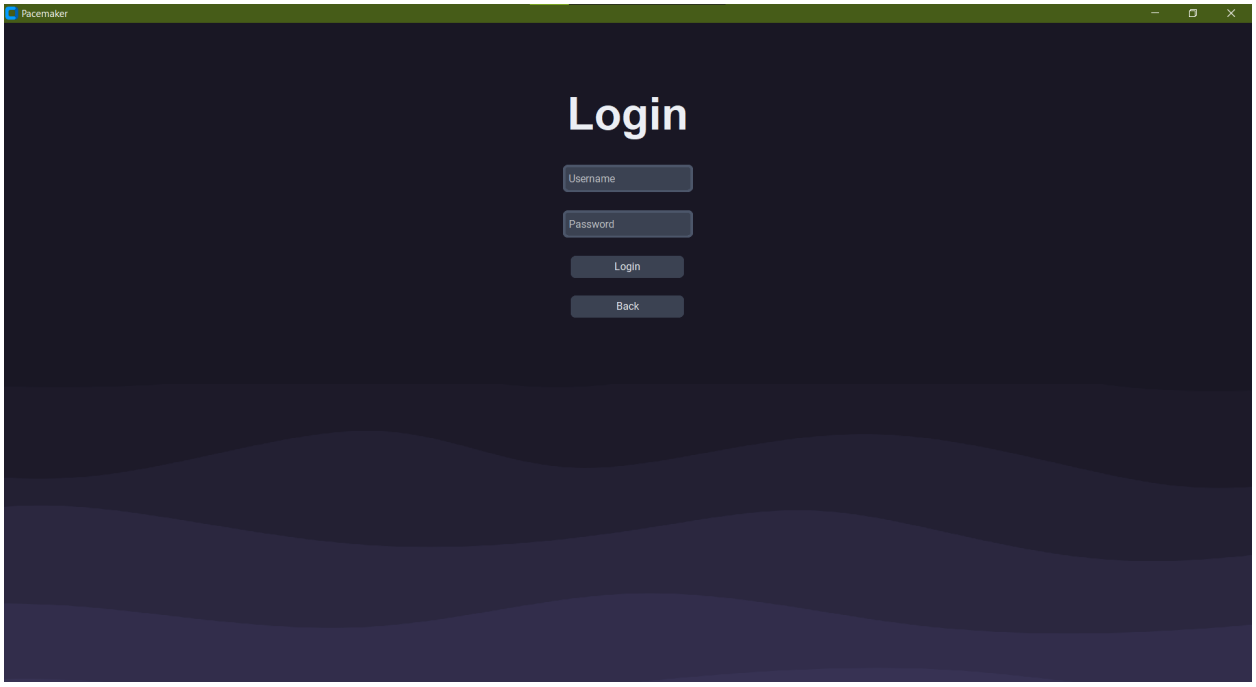


Figure 34. DCM - Finalized Login Screen

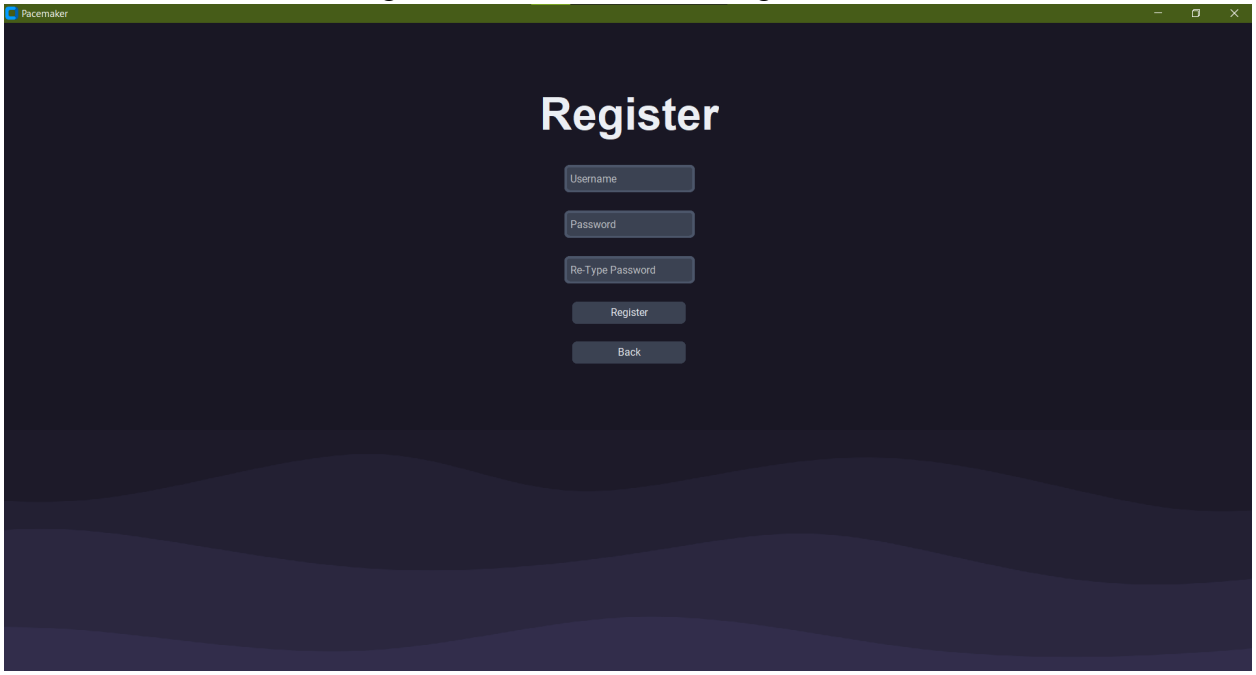


Figure 35. DCM - Account Registration Screen

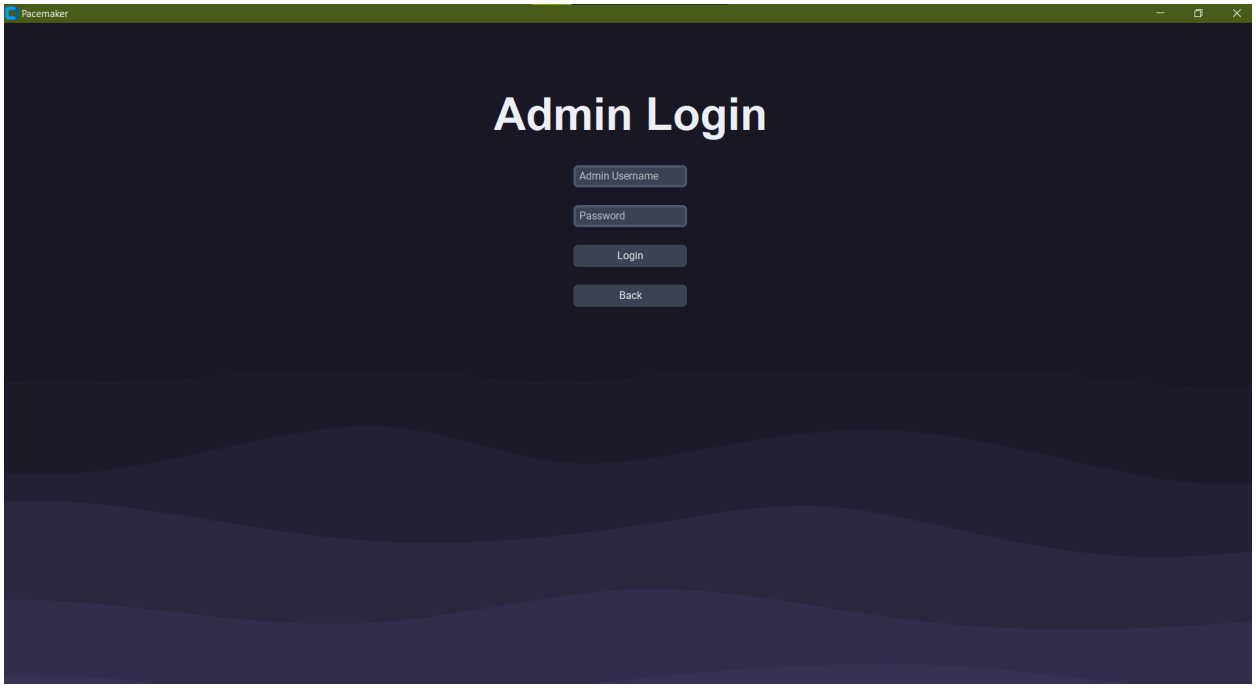


Figure 36. DCM - Admin Login Screen

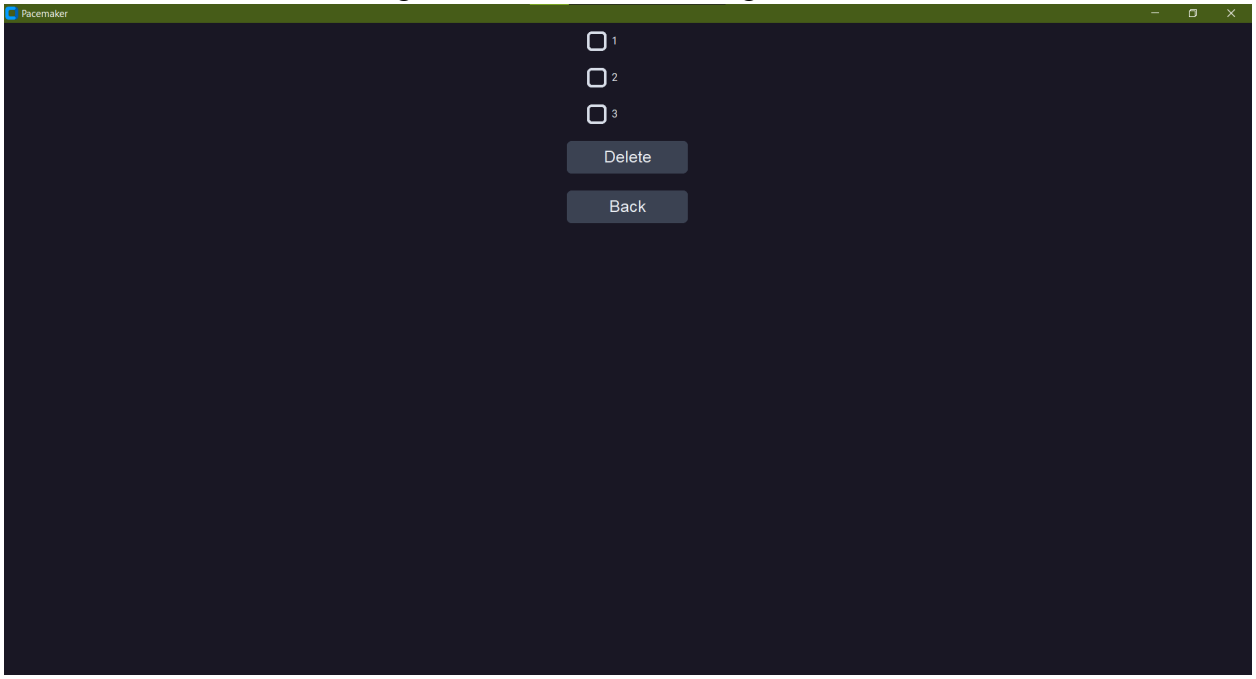


Figure 37. DCM - Main App Interface Connected Users Screen

Figure 38. DCM - Main App Interface ECG Output Screen

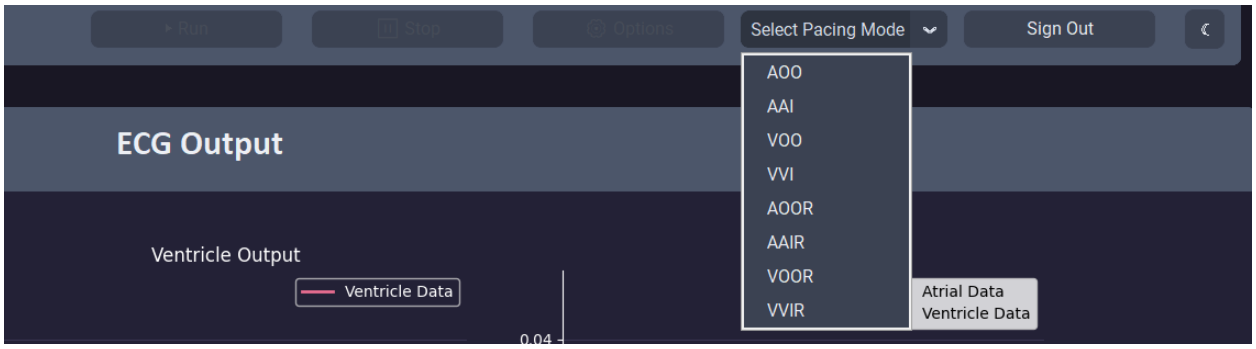


Figure 39. DCM - Mode Selection Screen

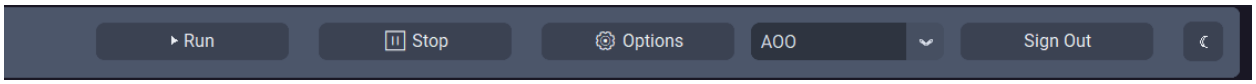


Figure 40. DCM - Finalized Button Options

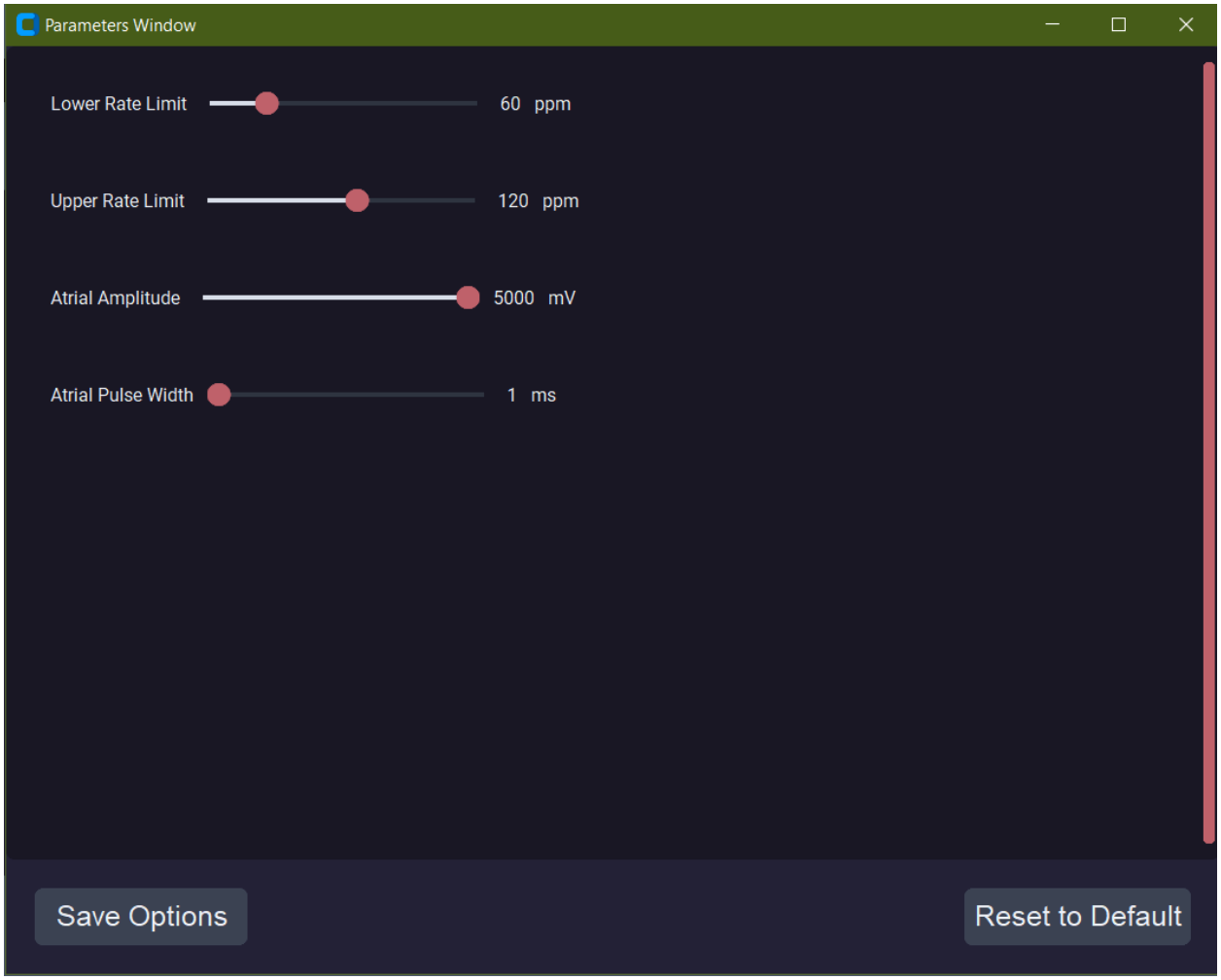


Figure 41. DCM - Finalized Parameters Modification Screen

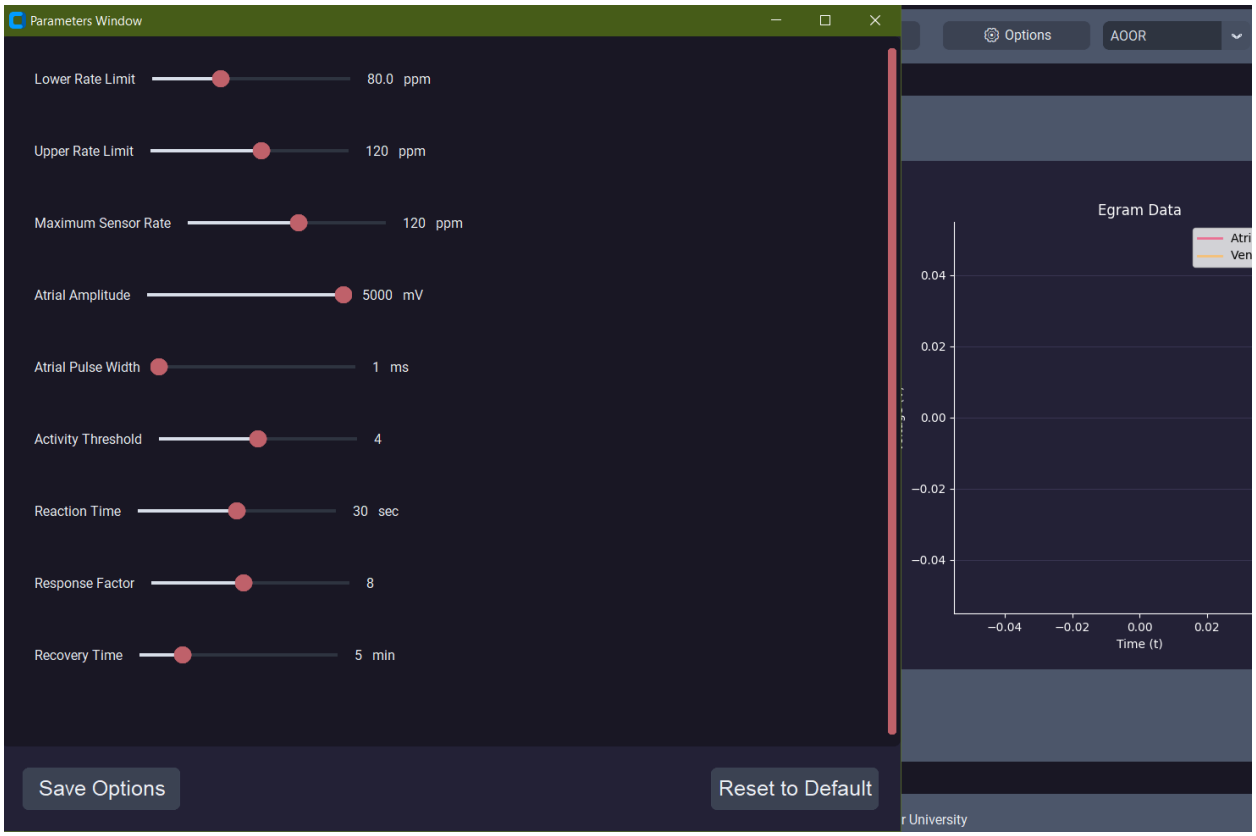


Figure 42. DCM - Finalized Parameters Modification Screen

Figure 43. DCM - Egram Plot Screen For AOO With A On and V On

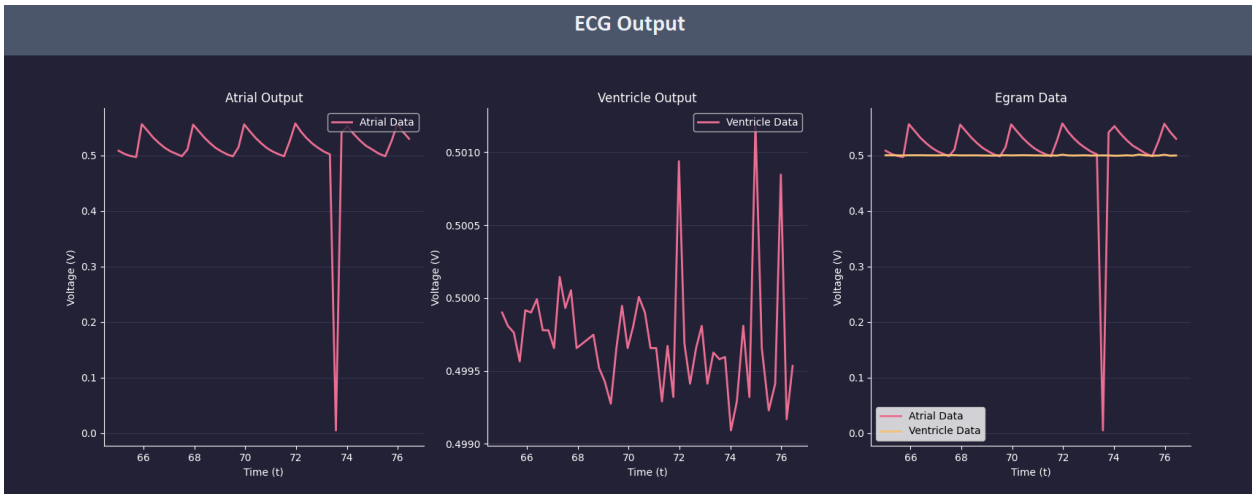


Figure 44. DCM - Egram Plot Screen For AOO With A On and V Off

Figure 45. DCM - Egram Plot Screen For AOO With A Off and V Off

Generated HTML Report

PACE++ Printed Report

Institution Name: McMaster University

Date and Time: 16:25:2023

Version: PACE++ V2.16.3

Serial Number: 000000123456

Table of Contents

Bradycardia Parameters

Parameter Name	Value	Unit
Mode	AOO	
Lower Rate Limit	60	ppm
Upper Rate Limit	120	ppm
Maximum Sensor Rate	120	ppm
AV Delay	15	
Atrial Amplitude	5.0	V
Ventricular Amplitude	5.0	V
Atrial Pulse Width	1	ms
Ventricular Pulse Width	1	ms
Atrial Sensitivity	4.0	mV
Ventricular Sensitivity	4.0	mV
Absolute Refractory Period	250	ms
Ventricular Refractory Period	320	ms
Post Ventricular Absolute Refractory Period	320	ms
Activity Threshold	4	
Reaction Time	30	sec
Response Factor	8	

Resulted Figures

The following three figures below are the resultant graphs taken at the time interval chosen when the "print report" button was pressed. Figure 1 depicts the egram data for the electrical acitivity found inside the atrium of the heart paced by the ATR_SIGNAL pin located at A0 on the board. Figure 2 depicts the egram daata for the electrical activatiy related to ventricular pacing done by the VENT_SIGNAL pin located at A1 on the board. From these two graphs, we can look at the resultant graph where we can combine the two to examine the relationship between atrial and ventricular pacing the pacemaker.

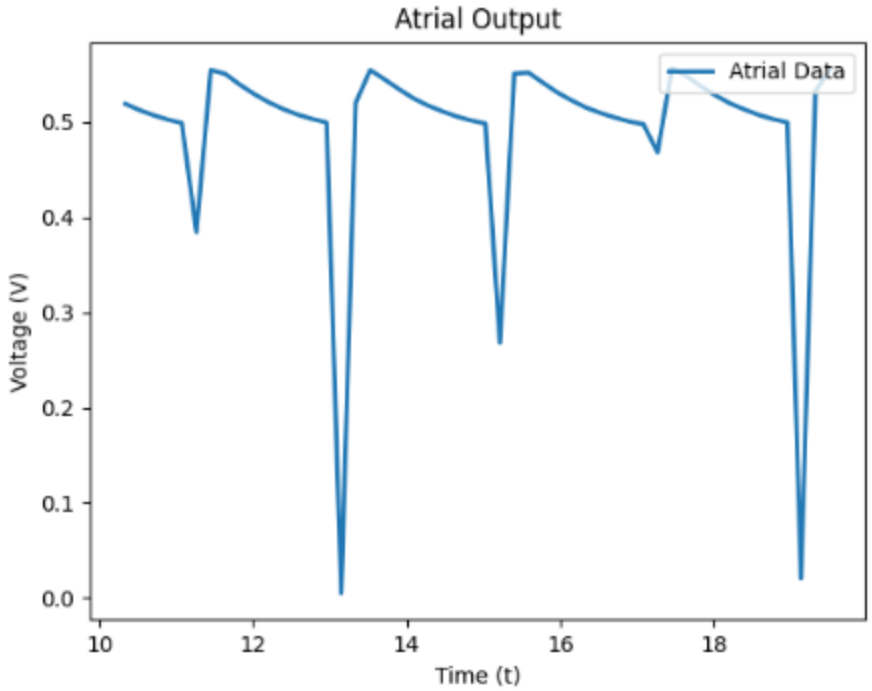


Figure 1: Atrial Output Graph

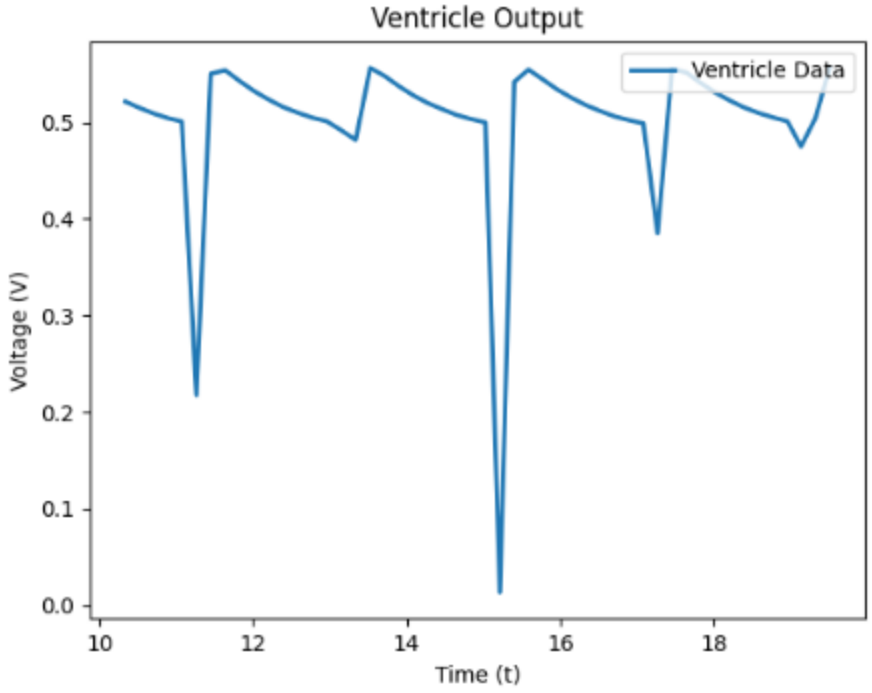


Figure 2: Ventricle Output Graph

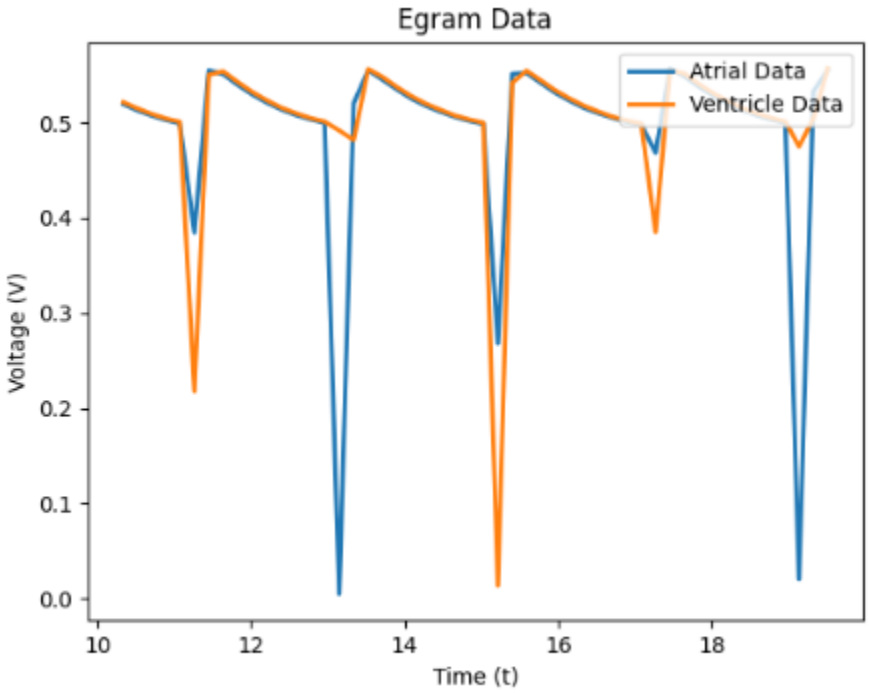


Figure 3: Combined Egram Data Graph

Monitored Variables & Constants

AOO

The AOO

Pin name	Function
D5_PACING_REF_PWM	Charges Primary Capacitor
D2_PACE_CHARGE_CTRL	Starts and Stops capacitor charging (c22)
D8_ATR_PACE_CTRL	Discharges the primary capacitor through the atrium
D10_PACE_GND_CTRL	Controls the switch directly following the tip
D11_ATR_GND_CTRL	Used when discharging the blocking capacitor through the atrium to allow no charge buildup
D4_Z_ATR_CTRL	Used to analyze impedance of the atrial electrode
D7_Z_VENT_CTRL	Same as z atrial ctrl but for ventricle
D12_VENT_GND_CTRL	Same as atrial ground ctrl but for ventricle
D9_VENT_PACE_CTRL	Same as atrial pace ctrl but for ventricle
Atrium_Pace_LED_State	On when pacing
Standby_Period	Standby period between pace and refractory states
Atrium_amplitude	Amplitude of paced signal
Atrium_pace_width	During of pace signal

VOO

Pin name	Function
D5_PACING_REF_PWM	Charges Primary Capacitor
D2_PACE_CHARGE_CTRL	Starts and Stops capacitor charging (c22)
D8_ATR_PACE_CTRL	Discharges the primary capacitor through the atrium
D10_PACE_GND_CTRL	Controls the switch directly following the tip
D11_ATR_GND_CTRL	Used when discharging the blocking capacitor through the atrium to allow no charge buildup
D4_Z_ATR_CTRL	Used to analyze impedance of the atrial electrode
D7_Z_VENT_CTRL	Same as z atrial ctrl but for ventricle
D12_VENT_GND_CTRL	Same as atrial ground ctrl but for ventricle
D9_VENT_PACE_CTRL	Same as atrial pace ctrl but for ventricle
Ventricle_Pace_LED_State	On when pacing
Standby_Period	Standby period between pace and refractory states
Ventricle_amplitude	Amplitude of paced signal
Ventricle_pace_width	During of pace signal

AAI

Pin name	Function
D5_PACING_REF_PWM	Charges Primary Capacitor
D2_PACE_CHARGE_CTRL	Starts and Stops capacitor charging (c22)
D8_ATR_PACE_CTRL	Discharges the primary capacitor through the atrium
D10_PACE_GND_CTRL	Controls the switch directly following the tip
D11_ATR_GND_CTRL	Used when discharging the blocking capacitor through the atrium to allow no charge buildup
D4_Z_ATR_CTRL	Used to analyze impedance of the atrial electrode
D7_Z_VENT_CTRL	Same as z atrial ctrl but for ventricle
D12_VENT_GND_CTRL	Same as atrial ground ctrl but for ventricle
D9_VENT_PACE_CTRL	Same as atrial pace ctrl but for ventricle

Atrium_Pace_LED_State	On when pacing
Standby_Period	Standby period between pace and refractory states
Atrium_Refractory_Period	Duration where no paces can be delivered after pacing
D0_ATR_CMP_DETECT	Detects pace in atrium
D6_ATR_CMP_REF_PWM	Compares pwm signal for sensing
Atrium_amplitude	Amplitude of paced signal
Atrium_pace_width	During of pace signal

VVI

Pin name	Function
D5_PACING_REF_PWM	Charges Primary Capacitor
D2_PACE_CHARGE_CTRL	Starts and Stops capacitor charging (c22)
D8_ATR_PACE_CTRL	Discharges the primary capacitor through the atrium
D10_PACE_GND_CTRL	Controls the switch directly following the tip
D11_ATR_GND_CTRL	Used when discharging the blocking capacitor through the atrium to allow no charge buildup
D4_Z_ATR_CTRL	Used to analyze impedance of the atrial electrode
D7_Z_VENT_CTRL	Same as z atrial ctrl but for ventricle
D12_VENT_GND_CTRL	Same as atrial ground ctrl but for ventricle
D9_VENT_PACE_CTRL	Same as atrial pace ctrl but for ventricle
Venticle_Pace_LED_State	On when pacing
Standby_Period	Standby period between pace and refractory states
Ventricle_Refractory_Period	Duration where no paces can be delivered after pacing
D1_VENT_CMP_DETECT	Detects pace in atrium
D3_VENT_CMP_REF_PWM	Compares pwm signal for sensing
Ventricle_amplitude	Amplitude of paced signal
Ventricle_pace_width	During of pace signal

Decision Tables

Pacemaker

AOO

	Pace	Refractory
[after(Standby_Period,msec)]	1	0
[after(Atrium_Pace_Width,msec)]	0	1

VOO

	Pace	Refractory
[after(Standby_Period,msec)]	1	0
[after(Ventricle_Pace_Width,msec)]	0	1

AAI

	Pace	Refractory	Junction
[after(Atrium_Pace_Width,msec)]	0	1	*
[after(Atrium_Refractory_Period,msec)]	*	0	1
after(Standby_Period,msec)	1	*	0
[D0_ATR_CMP_DETECT == 1]	0	1	0

VVI

	Pace	Refractory	Junction
[after(Ventricle_Pace_Width,msec)]	0	1	*
[after(Ventricle_Refractory_Period,msec)]	*	0	1
after(Standby_Period,msec)	1	*	0
[D1_VENT_CMP_DETECT == 1]	0	1	0

DCM

Conditions (Inputs):					
Login Attempt	*1*	*1*	*0*	*0*	*0*
Account Creation	*0*	*0*	*1*	*1*	*1*
Matching Credentials in Database	*1*				
Max Users in Database	*	*	*	*1*	*0*

Actions (Outputs):					
Login Granted	*1*	*0*	*0*	*0*	*1*
New Account Created	*0*	*0*	*0*	*0*	*1*

Conditions (Inputs):

VOO Mode	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*
VVI Mode	*0*	*0*	*1*	*1*	*0*	*0*	*0*	*0*
AOO Mode	*0*	*0*	*0*	*0*	*1*	*1*	*0*	*0*
AAI Mode	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*
VOO Parameter Changed	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*
VVI Parameter Changed	*	*	*	*	*	*	*	*
AOO Parameter Changed	*	*	*	*	*	*	*	*
AAI Parameter Changed	*	*	*	*	*	*	*	*

Actions (Outputs):								
Change Allowed	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*0*

Software Testing/Assurance Cases

Assurance Cases:

Claims		Context and Assumptions	Strategy and Argument	Evidence
1	The pacemaker and DCM are a cohesive device that is safe to use by both patients, technicians, and doctors.	The pacemaker and DCM device meet the requirements outlined in section 1 of the documentation.	N/A	*See below*
1.1	The DCM is safe and secure to use	*See below.*	*See below.*	*See below*
1.1.1	The user login and account creation is without fault	Users are allowed to register accounts and login. Registering should follow the general standard of creating an account with a username and password. The login function allows for the user to login only with the correct username and password.	Strategy: Test all possible combinations of inputs: blank, incorrect, and correct and check if they return the expected value. Argument: If all the expected results occur on the login/register screen, then the user login and account has been created correctly.	*See DCM Test Cases 1-6*
1.1.2	The user data is saved on the pacemaker	As parameter data is saved locally on the device, the data should be saved under each individual user correctly.	Strategy: Check if user's saved parameters for each mode automatically populate when the user logs in. Argument: If the correct values show up in the parameters window, then the data is saved correctly.	*See Figure X. and Figure X. DCM Test Case 8.*
1.1.2.1	User data is encrypted in the event of a data leak	As saved parameter data is stored locally on the device, it is important to make sure that in the event of a data leak user data is not lost.	Strategy: Open user_accounts file and try to get usernames and passwords. Argument: If you can't figure out how to login using the file that stores it, then the accounts are secure.	*See Figure X. and Figure X. in DCM Test Case 8.*

1.1.3	The admin data is safe and is able to be used to edit user data	Since there is a limit of 10 users on the pacemaker, if we want to be able to scale up the software. There should be the ability to remove old users and add new users with unique parameter data.	Strategy: Log in with admin account and try deleting a patient's account. Argument: If the patient is deleted then the admin is able to edit profiles and different accounts.	*See DCM Test Case 9*
1.1.4	The GUI does not allow for incorrect input of parameter data	Since each input parameter relates to a specific function of the pacemaker that can affect the heart of a user. Incorrect parameter data should not be allowed to be entered into the GUI.	Strategy: Try dragging a slider outside the range or to an intermediary value that isn't supported. Argument: If the slider is restricted to valid values, than the user can't input incorrect values.	*See DCM Test Case 10*
1.1.5	The DCM has a clear separation between user data	User data should be specific to each user, as varying cardiovascular diseases should have difference pacing parameter requirements. The DCM separates user data and makes sure not to mix data between each other.	Strategy: Log into a user, change the parameters for a specific mode and save that configuration. Then log into another new user, select the same mode and see if the parameters used are the same. Argument: The second account is new so it should have the default values. If they are the default values and not the values saved in the first account the users have clear separation.	*See Figure X. and Figure X. DCM Test Case 8*
1.2	The pacemaker and DCM can safely communicate and accurately between each other	The parameter data that is sent and received from the DCM to the pacemaker should be identical. It is important to verify that the data being sent is the data being received. Pyserial allows for the clear separation of reading and writing data.	*See below.*	*See Figure X. in DCM Test Case 7 demonstrating that the data packet containing the parameters was successfully sent* *See Serial Test Cases: ID 6*
1.2.1	The serial communication	When the write function is called the data sent is only	Strategy: Change the parameters and click save. If	*See Serial Test Cases: ID 2*

	should only write when it is supposed to write	written to the pacemaker.	the user logs out and logs back in the parameters for that mode should be the saved ones. Argument: If they are the saved values, then the pacemaker was able to write correctly.	
1.2.2	The serial communication should only read when it is supposed to read	When the read function is called the data it returns should only be read.	Strategy: Change the parameters and click save. If the user logs out and logs back in the parameters for that mode should be the saved ones. Argument: If they are the saved values, then the pacemaker was able to read correctly.	*See Serial Test Cases: ID 3*
1.2.3	The serial communication should only pull egram data when it is supposed during graph creation	When the egram data is called through the DCM, the only data that should be returned is the egram data.	Strategy: The egram graph should not be noisy. Argument: If the graph has random values, then it's getting data from somewhere random.	*See Serial Test Cases: ID 4*
1.2.4	The pull rate of the graph should be adequate to see full graph data	The connection between the pacemaker and the DCM should be fast enough to draw a complete graph that should have enough data that allows for diagnosis to be made.	Strategy: The egram graph should be periodic. Argument: If not, the data is not transmitting correctly or fast enough for a smooth graph.	*See Serial Test Cases: ID 5*
2	The pacemaker adequately implements all modes for user	All modes meet the requirements outlined in section 1 of the documentation.		
2.1	AOO is implemented correctly	All parameters relating to mode can be sent using serial communication. The pacing of the pacemaker can be viewed with heartview.	Strategy: Send different values for AOO parameters and monitor pacing on heartview Argument: If pacing is monitored to be continuous	*See AOO test case 1 and 2*

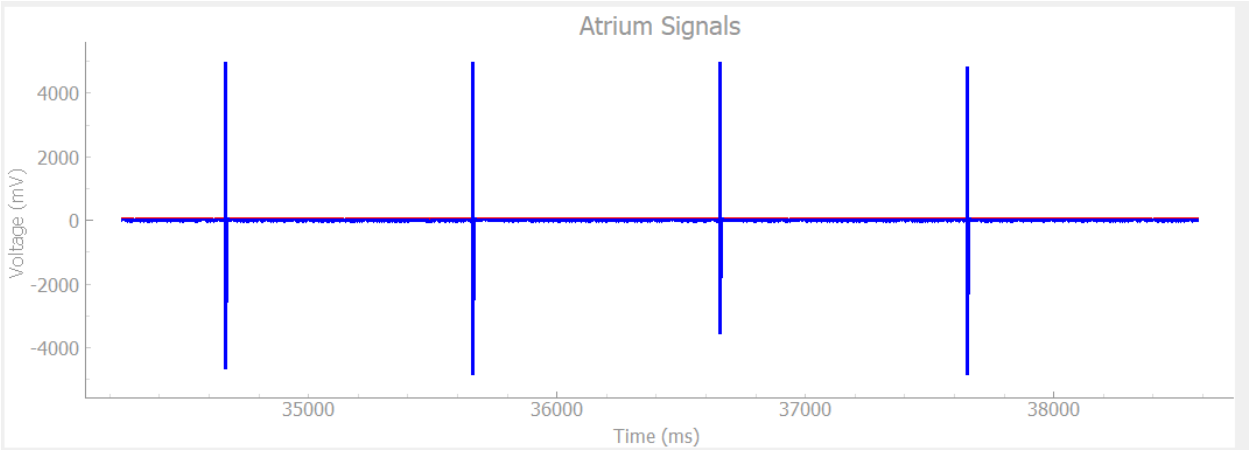
			and showing the sent amplitude/LRL/PulseWidth, then the mode can be confirmed to be functional	
2.2	VOO is implemented correctly	All parameters relating to mode can be sent using serial communication. The pacing of the pacemaker can be viewed with heartview.	Strategy: Send different values for VOO parameters and monitor pacing on heartview Argument: If pacing is monitored to be continuous and showing the sent amplitude/LRL/PulseWdith, then the mode can be confirmed to be functional	*See VOO test case 1 and 2*
2.3	AAI is implemented correctly	All parameters relating to mode can be sent using serial communication. The pacing of the pacemaker can be viewed with heartview.	Strategy: Set the natural heartbeat to different values to see if the pacemaker is pulsing in between on missed beats Argument: If AAI paces on missed beats, it is correctly sensing and pacing.	*See AAI test case 1, 2, 3*
2.3	VVI is implemented correctly	All parameters relating to mode can be sent using serial communication. The pacing of the pacemaker can be viewed with heartview.	Strategy: Set the natural heartbeat to different values to see if the pacemaker is pulsing in between on missed beats Argument: If VVI paces on missed beats, it is correctly sensing and pacing.	*See VVI test case 1, 2, 3*
2.4	AOOR, VOOR, AAIR, VVIR are implemented correctly	All parameters relating to mode can be sent using serial communication. The pacing of the pacemaker can be viewed with heartview.	Strategy: Shake the pacemaker and see if pacing increases Argument: If pulses speed up proportional to the amount of shaking, rate adaptive is functional	*See Rate adaptive test cases 1-8*

Pacemaker Cases

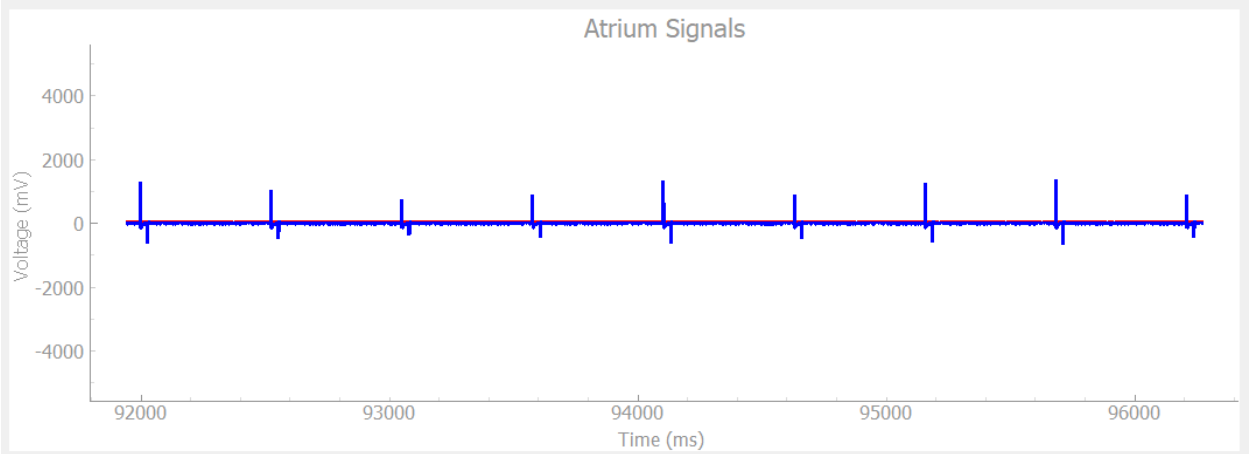
Testing the pacemaker state flow was completed by running each mode under several different heart conditions using Heartview. The signals that were used to conclude the state as functional are included below. The pacemaker shield's physical LED was also used for additional testing to ensure that the pacemaker was actively pacing.

The following test cases verify the correct operation of the pacemaker stateflow.

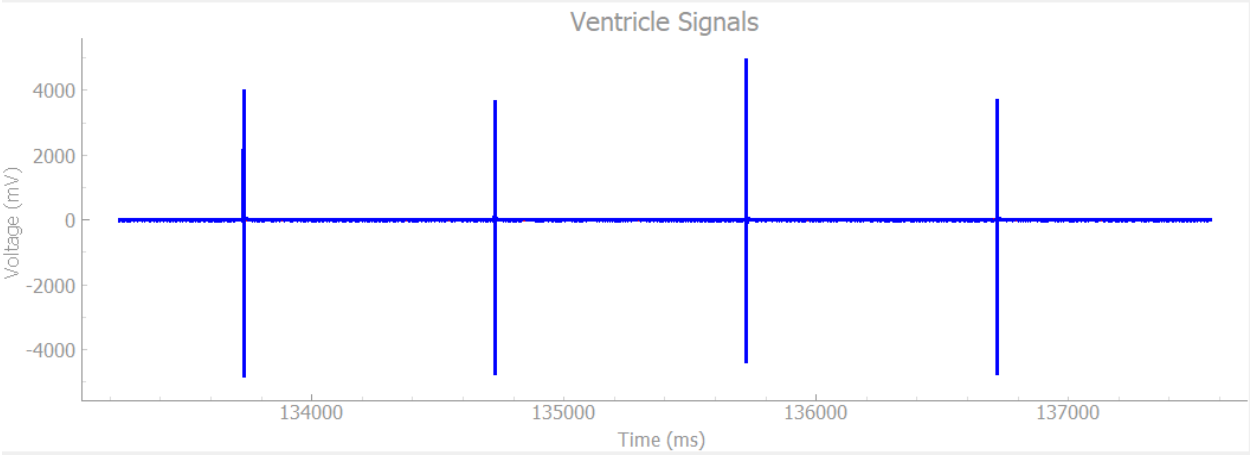
AOO Case 1: LRL=60, AAmp=5, APulseWidth=1



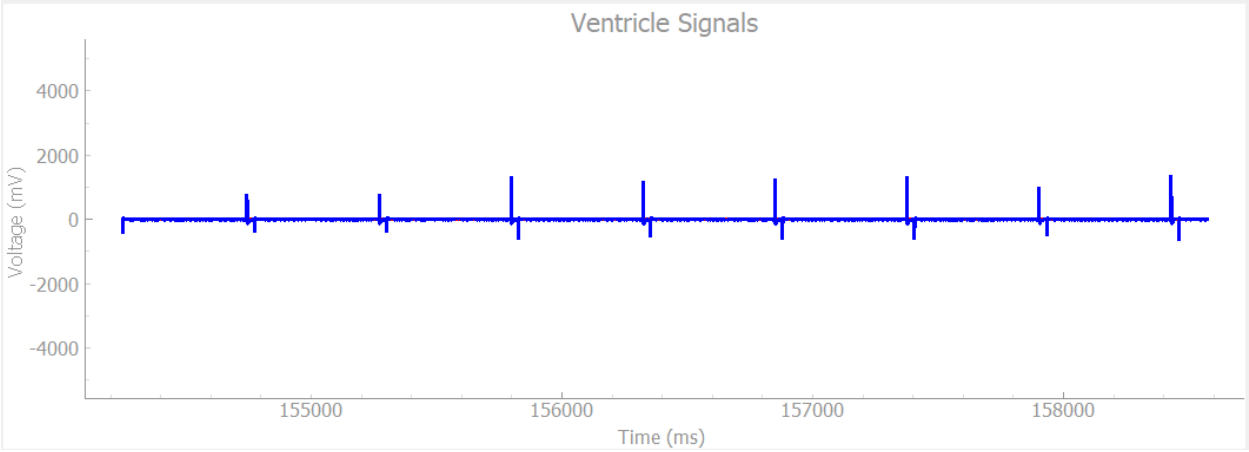
AOO Case 2: LRL=120, AAmp=1, APulseWidth=15



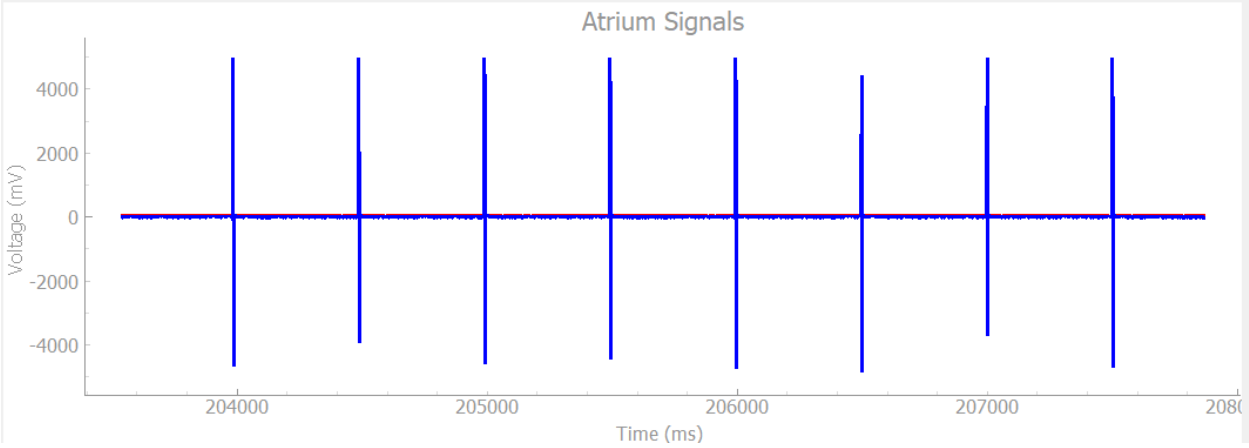
VOO Case 1: LRL=60, VAmp=5, VPulseWidth=1



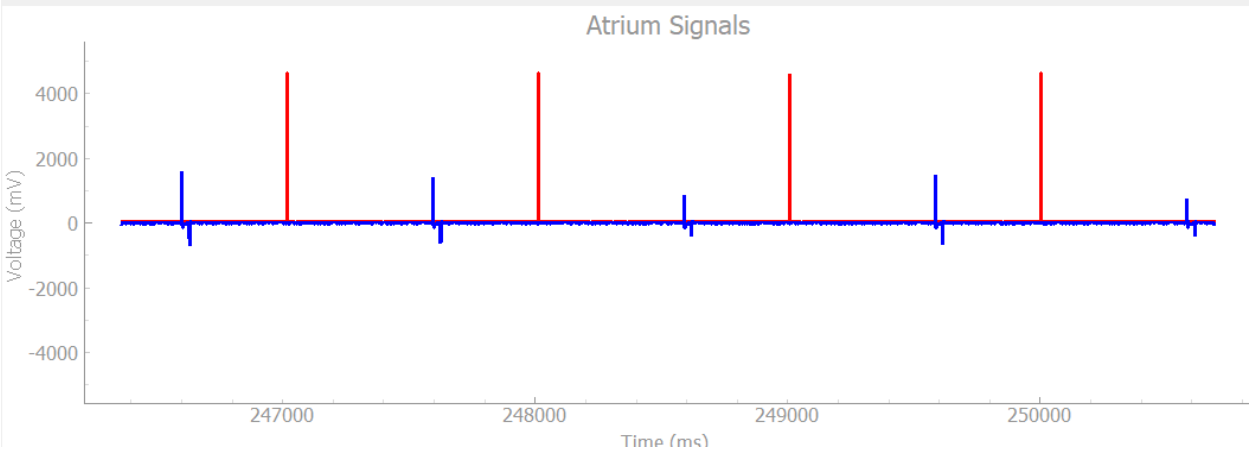
VOO Case 2: LRL=120, VAmp=1, VPulseWidth=15



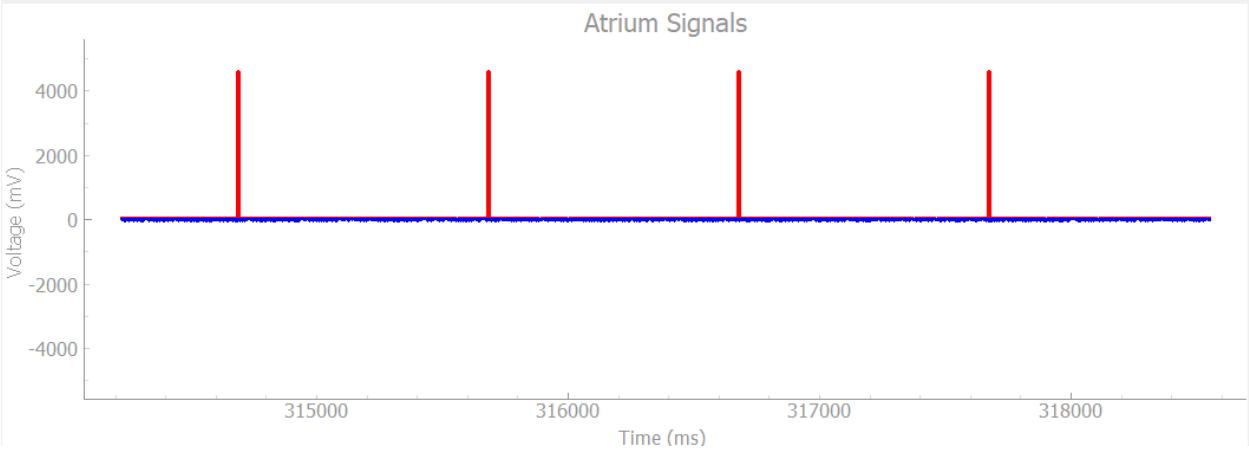
AAI Case 1: LRL=120, AAmp=5, APulseWidth=1, No Natural Pulse



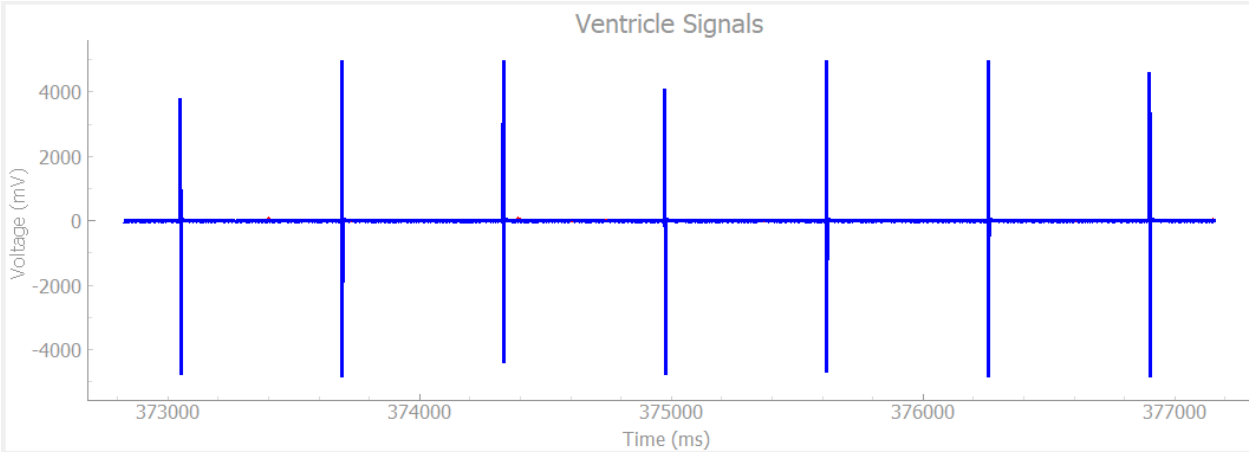
AAI Case 2: LRL=120, AAmp=1, APulseWidth=15, Natural Pulse = 60



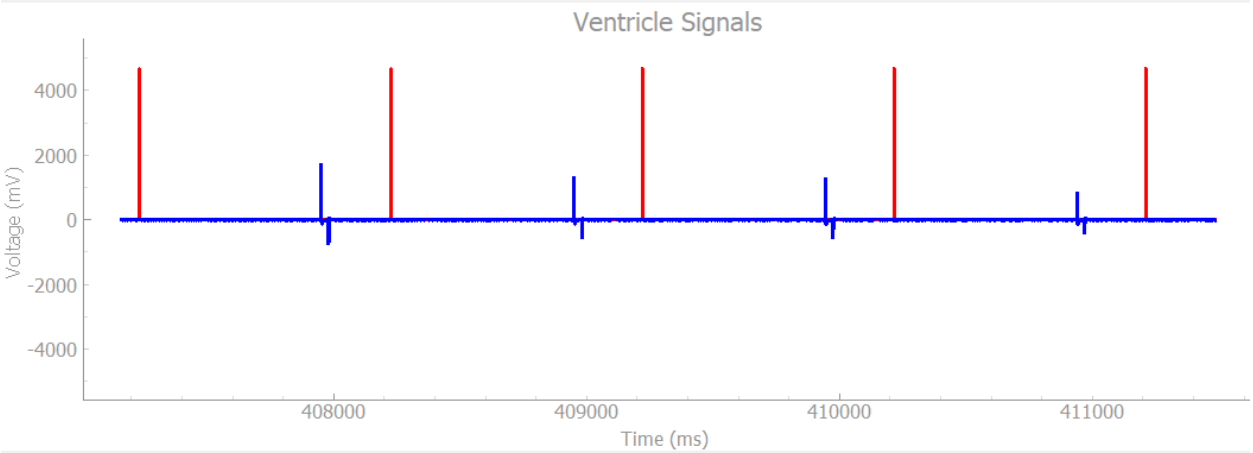
AAI Case 3: LRL=60, AAmp=5, APulseWidth=1, Natural Pulse = 60



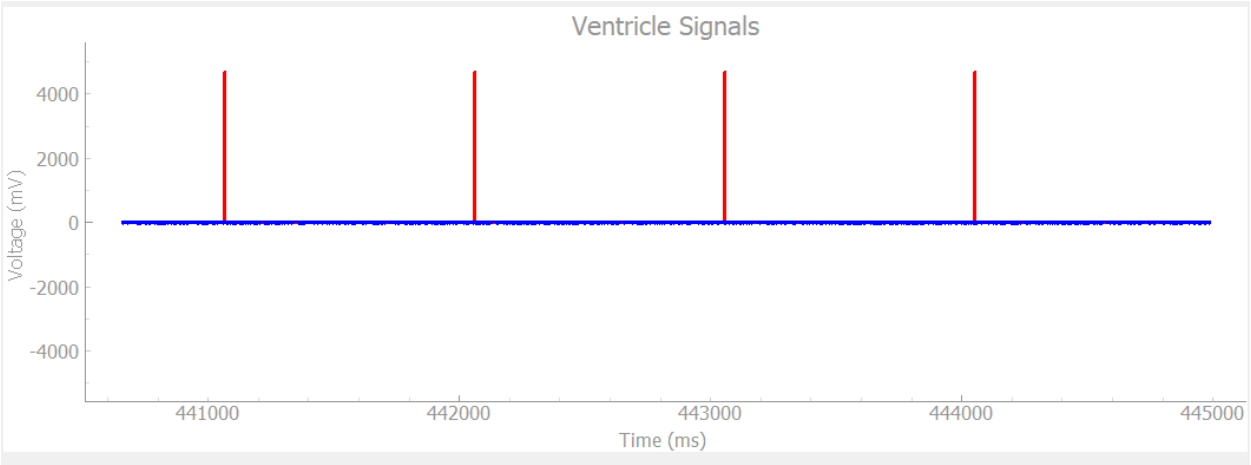
VVI Case 1: LRL=120, VAmp=5, VPulseWidth=1, No Natural Pulse



VVI Case 2: LRL=120, VAmp=1, VPulseWidth=15, Natural Pulse = 60



VVI Case 3: LRL=60, VAmp=5, VPulseWidth=1, Natural Pulse = 60



Rate Adaptive

The following test cases verify the correct function of the rate adaptive subsystem.

Test Case 1: Non Rate Adaptive Mode

Test Justification and Purpose
The purpose of this test is to determine whether the Target_Rate will default to the LRL if we are in a non rate adaptive mode.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail

1	Non Rate Adaptive Mode	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 8 **Reaction Time:** 30 **Recovery Time:** 5 **Rate Adaptive On/Off:** 0 **LRL:** 60 **URL:** 120 **MSR:** 120	Target Rate = 60	Target Rate = 60	Pass

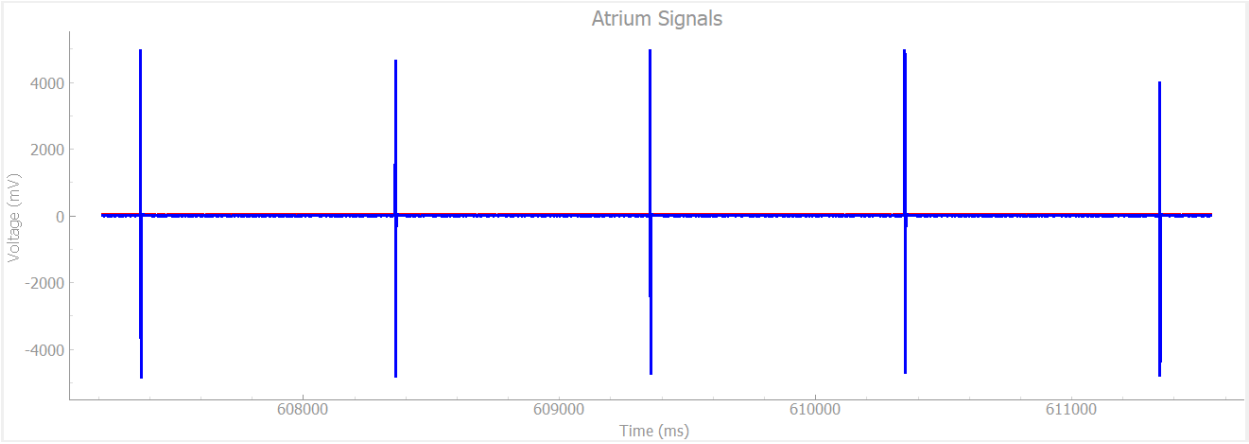


Figure 46. Before Shaking

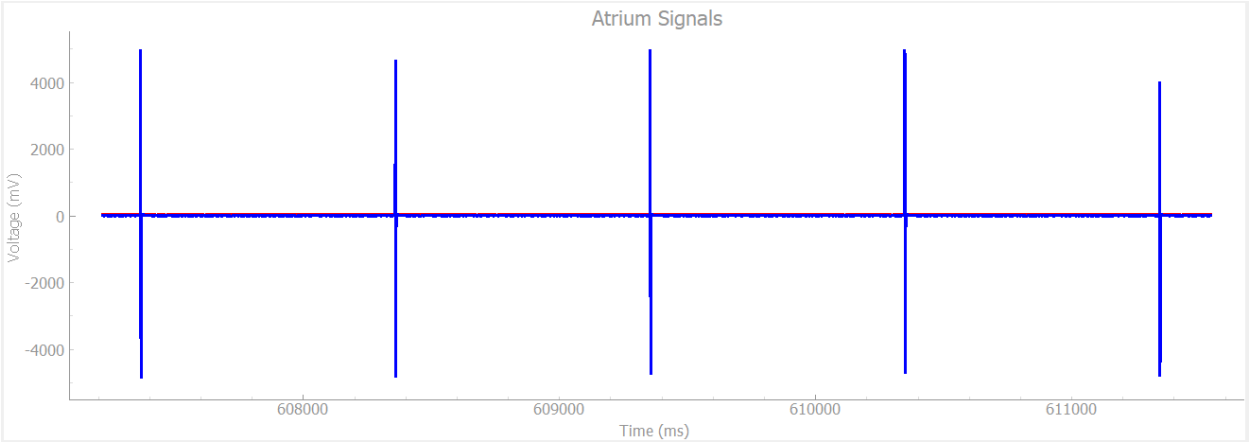


Figure 47. After Shaking

Test Case 2: Rate Adaptive Mode, Default Params

Test Justification and Purpose

The purpose of this test is to determine whether the Target_Rate will increase/decrease according to the activity measured by the accelerometer.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
2	Rate Adaptive Mode, Default Params	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 8 **Reaction Time:** 30 **Recovery Time:** 5 **Rate Adaptive On/Off:** 1 **LRL:** 60 **URL:** 120 **MSR:** 120	Target Rate increases to MSR as the board is shaken, Target Rate decreases to LRL once the board is put down	Target Rate increases to MSR as the board is shaken, Target Rate decreases to LRL once the board is put down	Pass

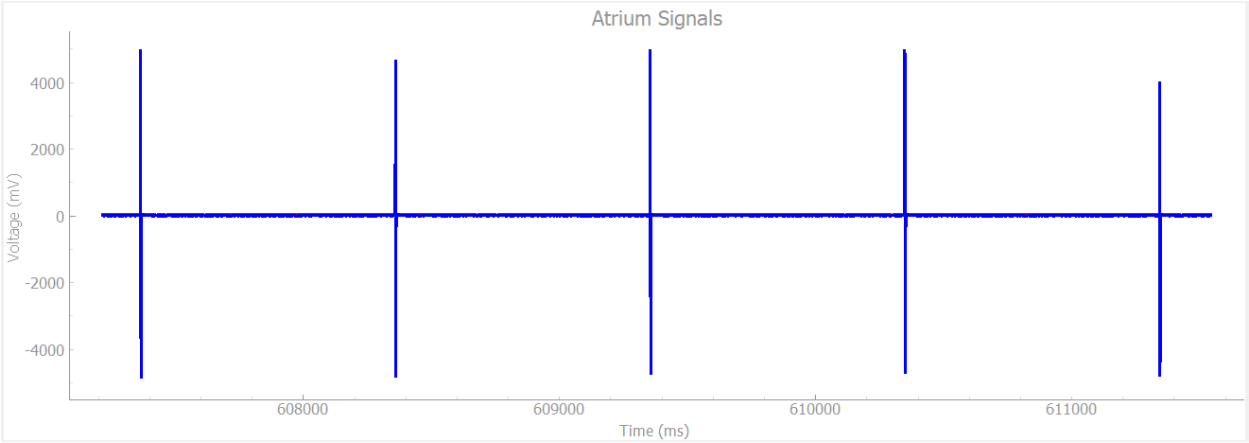


Figure 48. Before Shaking

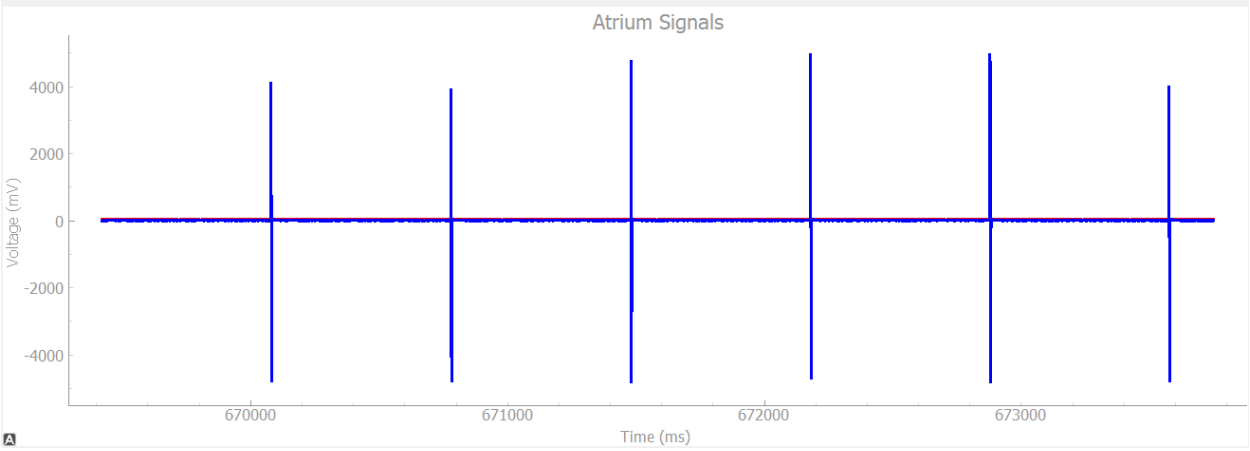


Figure 49. After Shaking

Test Case 3: Rate Adaptive Mode, MSR=LRL

Test Justification and Purpose
The purpose of this test is to determine whether the Target_Rate will cap at the MSR, even if the LRL and MSR are equal

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
3	Rate Adaptive Mode, MSR = LRL	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 8 **Reaction Time:** 30 **Recovery Time:** 5 **Rate Adaptive On/Off:** 1 **LRL:** 60 **URL:** 120 **MSR:** 60	Target Rate = 60	Target Rate = 60	Pass

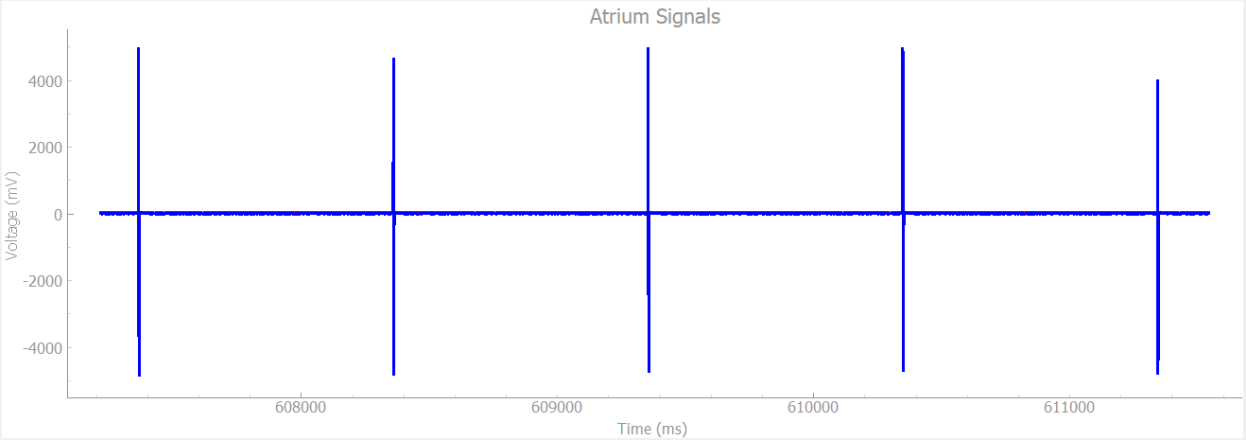


Figure 50. Before Shaking

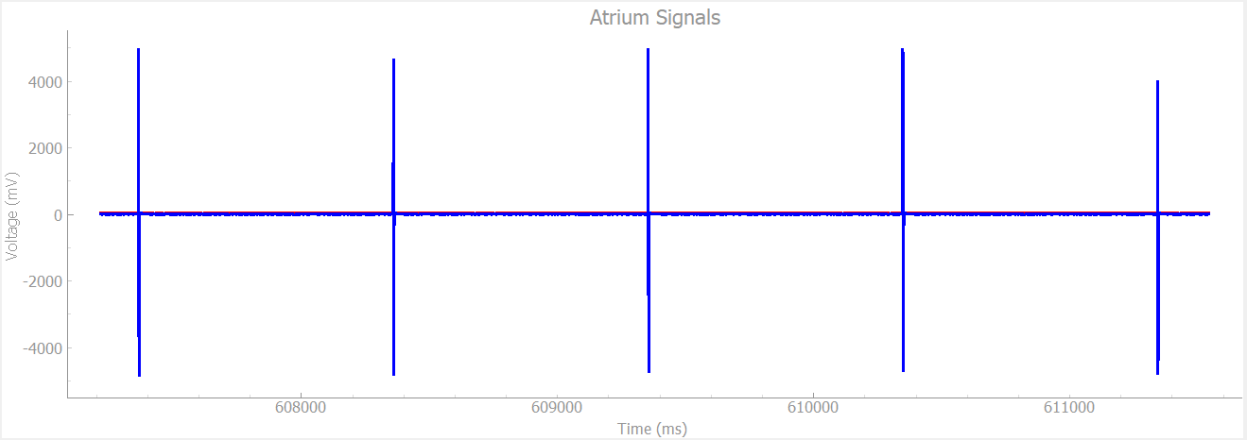


Figure 51. After Shaking

Test Case 4: Rate Adaptive Mode, MSR>URL

Test Justification and Purpose
The purpose of this test is to determine whether MSR will be lowered to to the URL if the URL is set lower than the MSR

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
4	Rate Adaptive Mode, MSR > URL	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 8 **Reaction Time:** 30 **Recovery Time:** 5 **Rate Adaptive On/Off:** 1 **LRL:** 60 **URL:** 90 **MSR:** 120	Target Rate caps at 90 once the board is shaken for long enough.	Target Rate caps at 90 once the board is shaken for long enough.	Pass

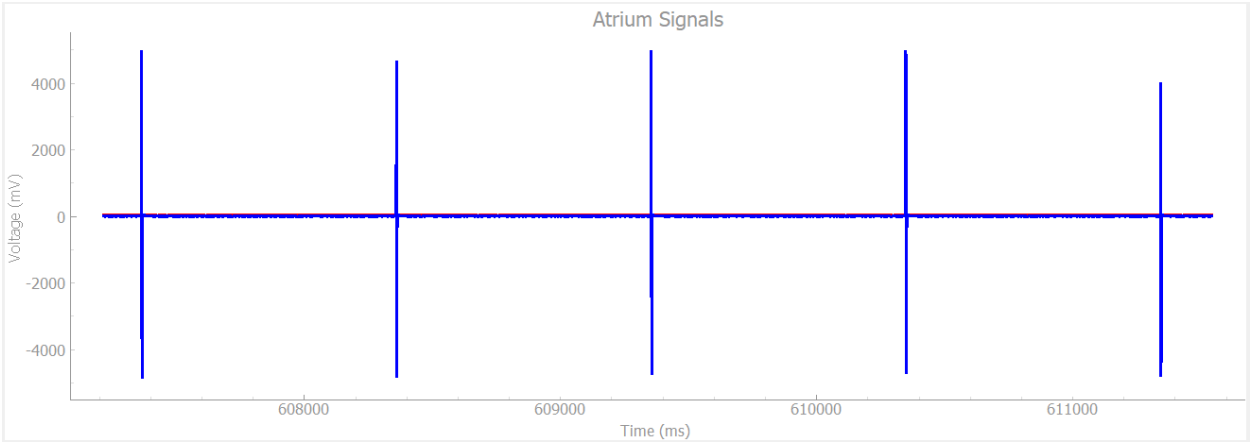
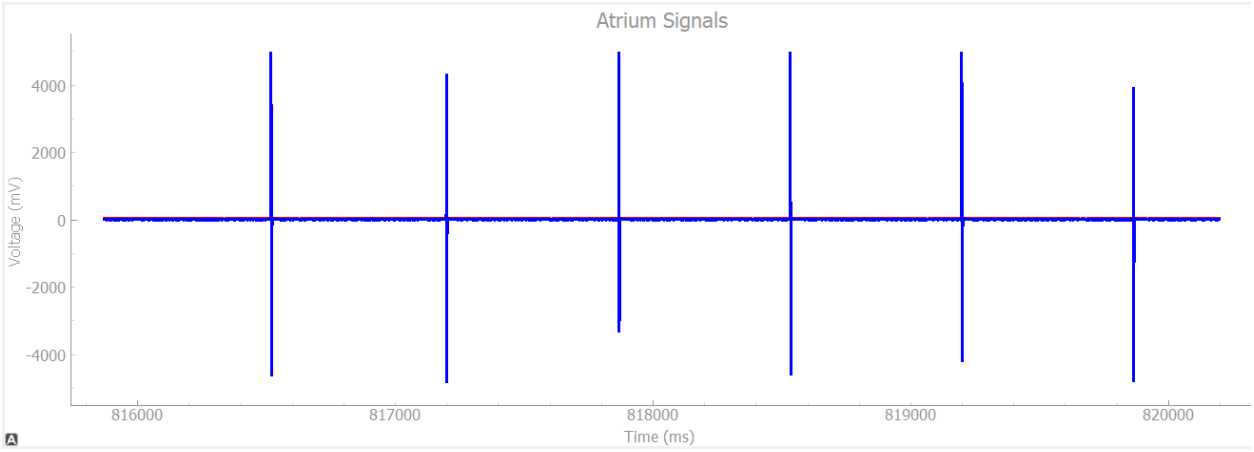


Figure 52. Before Shaking



Test Case 5: Rate Adaptive Mode, Changing Activity Threshold

Test Justification and Purpose
The purpose of this test is to determine whether altering the Activity Threshold will change how the Target Rate increases/decreases.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
5	Rate Adaptive Mode, Changing Activity Threshold	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10, 40 **Response Factor:** 8 **Reaction Time:** 30 **Recovery Time:** 5 **Rate Adaptive On/Off:** 1	More activity is necessary for the Target Rate to start increasing at higher activity thresholds, and will increase slower. Less	More activity is necessary for the Target Rate to start increasing at higher activity thresholds, and will increase slower. Less	Pass

		LRL: 60 **URL:** 120 **MSR:** 120	activity is necessary for the Target Rate to start increasing at lower activity thresholds, and will increase faster.	activity is necessary for the Target Rate to start increasing at lower activity thresholds, and will increase faster.	

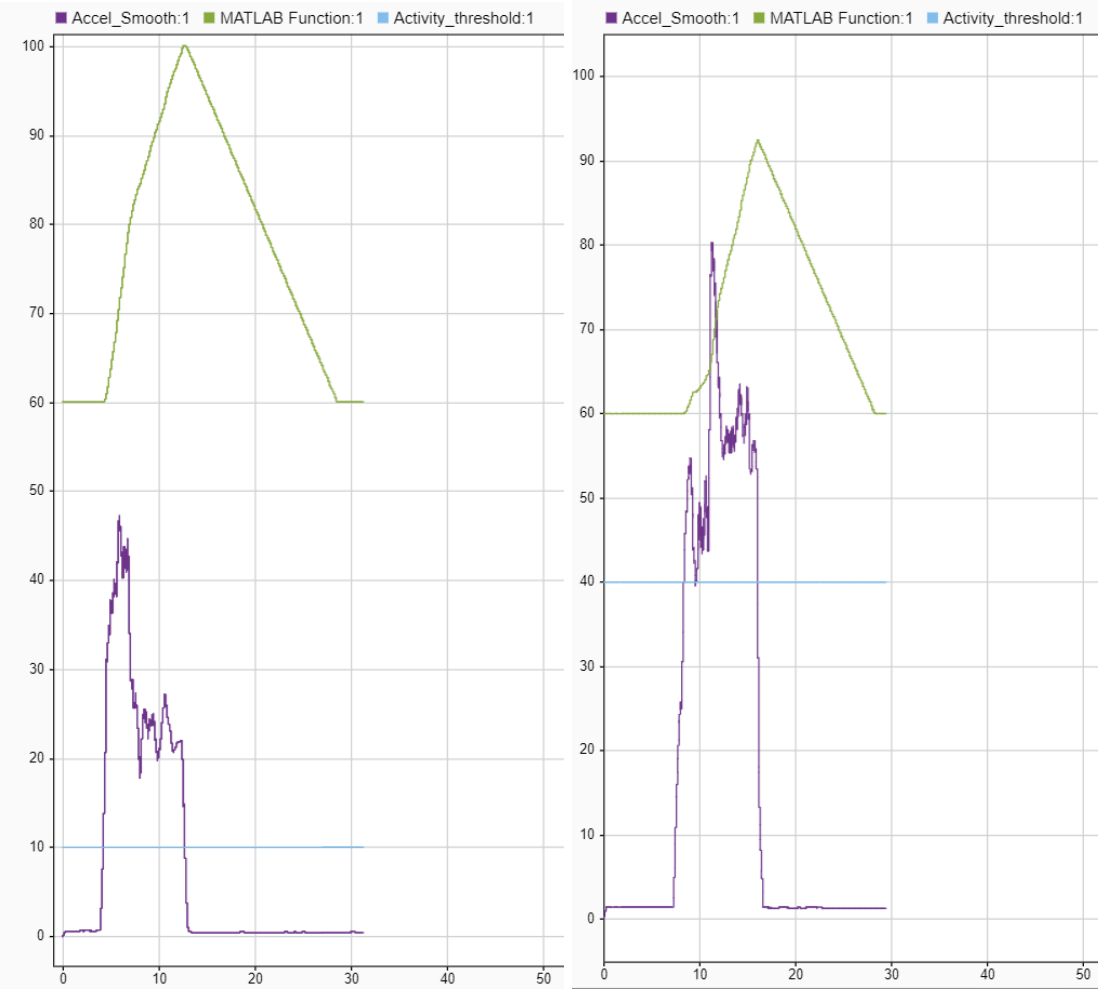


Figure 53. Changing Activity Threshold (Blue), Target Rate(Green), Activity (Purple)

Test Case 6: Rate Adaptive Mode, Changing Response Factor

Test Justification and Purpose

The purpose of this test is to determine whether altering the Response factor will change how the Target Rate increases/decreases.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
6	Rate Adaptive Mode, Changing Response Factor	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 1**,**16 **Reaction Time:** 30 **Recovery Time:** 5 **Rate Adaptive On/Off:** 1 **LRL:** 60 **URL:** 120 **MSR:** 120	The target rate increases/ decreases faster for the same amount of activity with higher response factor values. The target rate increases/ decreases slower for the same amount of activity with lower response factor values.	The target rate increases/ decreases faster for the same amount of activity with higher response factor values. The target rate increases/ decreases slower for the same amount of activity with lower response factor values.	Pass

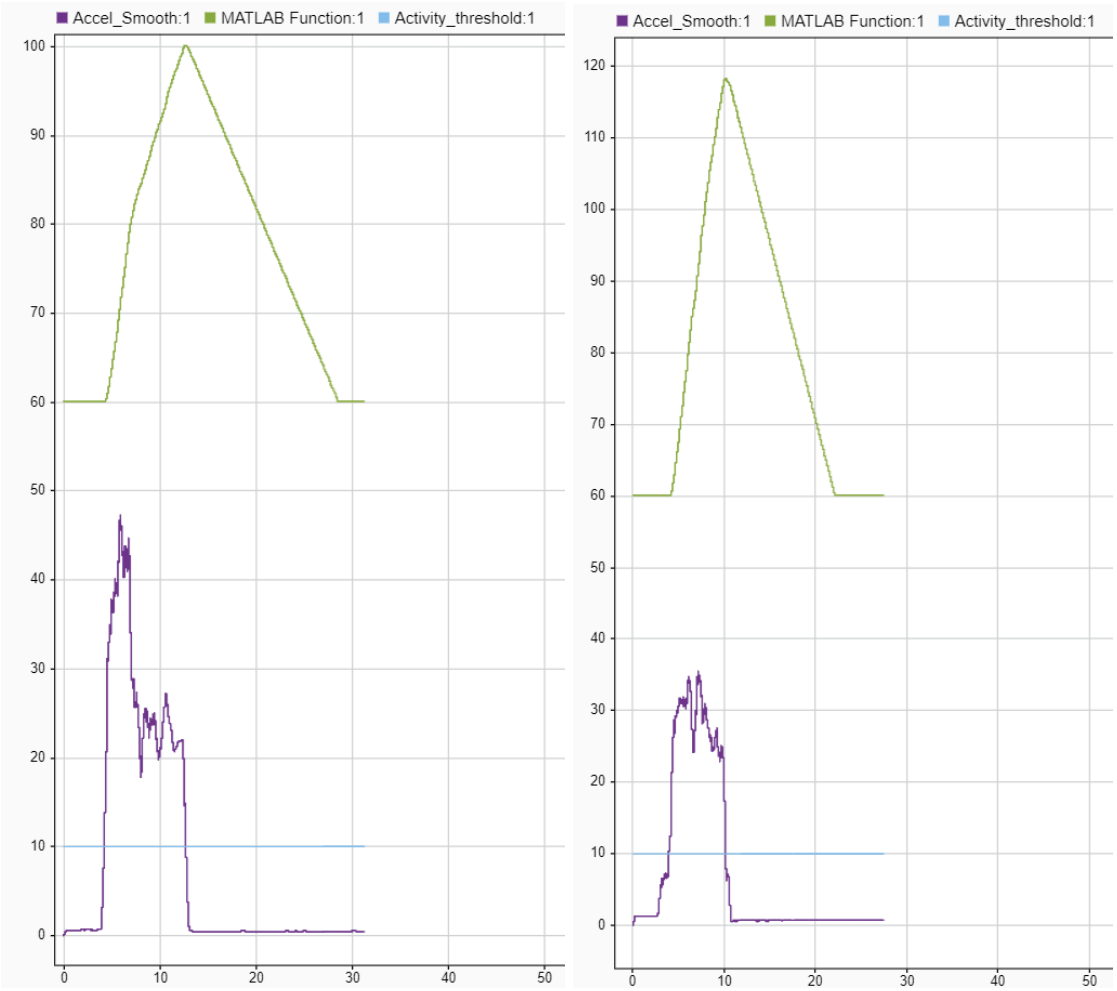


Figure 54. Changing Response Factor, Target Rate(Green), Activity (Purple)
Response Factor 8 (Left), Response Factor 16 (Right)

Test Case 7: Rate Adaptive Mode, Changing Reaction Time

Test Justification and Purpose
The purpose of this test is to determine whether altering the Reaction Time will change how the Target Rate increases/decreases.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
7	Rate Adaptive Mode, Changing Reaction	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 8	The target rate increases faster for lower reaction time values. The	The target rate increases faster for lower reaction time values. The	Pass

	Time	**Reaction Time:** 30,60 **Recovery Time:** 5 **Rate Adaptive On/Off:** 1 **LRL:** 60 **URL:** 120 **MSR:** 120	target rate increases slower for higher reaction time.	target rate increases slower for higher reaction time.	

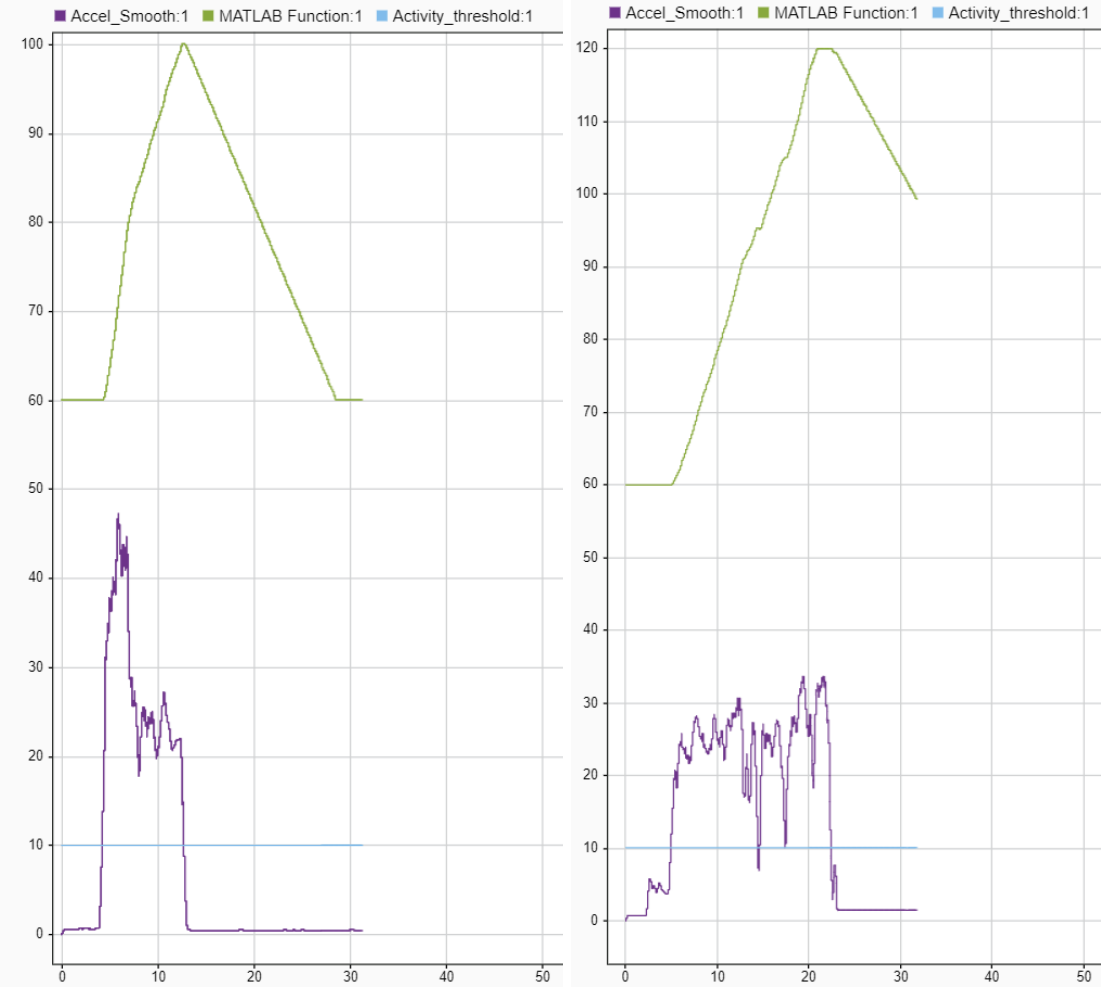


Figure 55. Changing Reaction Time, Target Rate(Green), Activity (Purple)
Reaction Time 30 (Left), Reaction Time 60 (Right)

Test Case 8: Rate Adaptive Mode, Changing Recovery Time

Test Justification and Purpose

The purpose of this test is to determine whether altering the Recovery Time will change how the Target Rate increases/decreases.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
8	Rate Adaptive Mode, Changing Reaction Time	**Activity:** Variable dependent on accelerometer data **Activity Threshold:** 10 **Response Factor:** 8 **Reaction Time:** 30 **Recovery Time:** 1,5 **Rate Adaptive On/Off:** 1 **LRL:** 60 **URL:** 120 **MSR:** 120	The target rate decreases faster for lower recovery time values. The target rate decreases slower for higher recovery time values.	The target rate decreases faster for lower recovery time values. The target rate decreases slower for higher recovery time values.	Pass

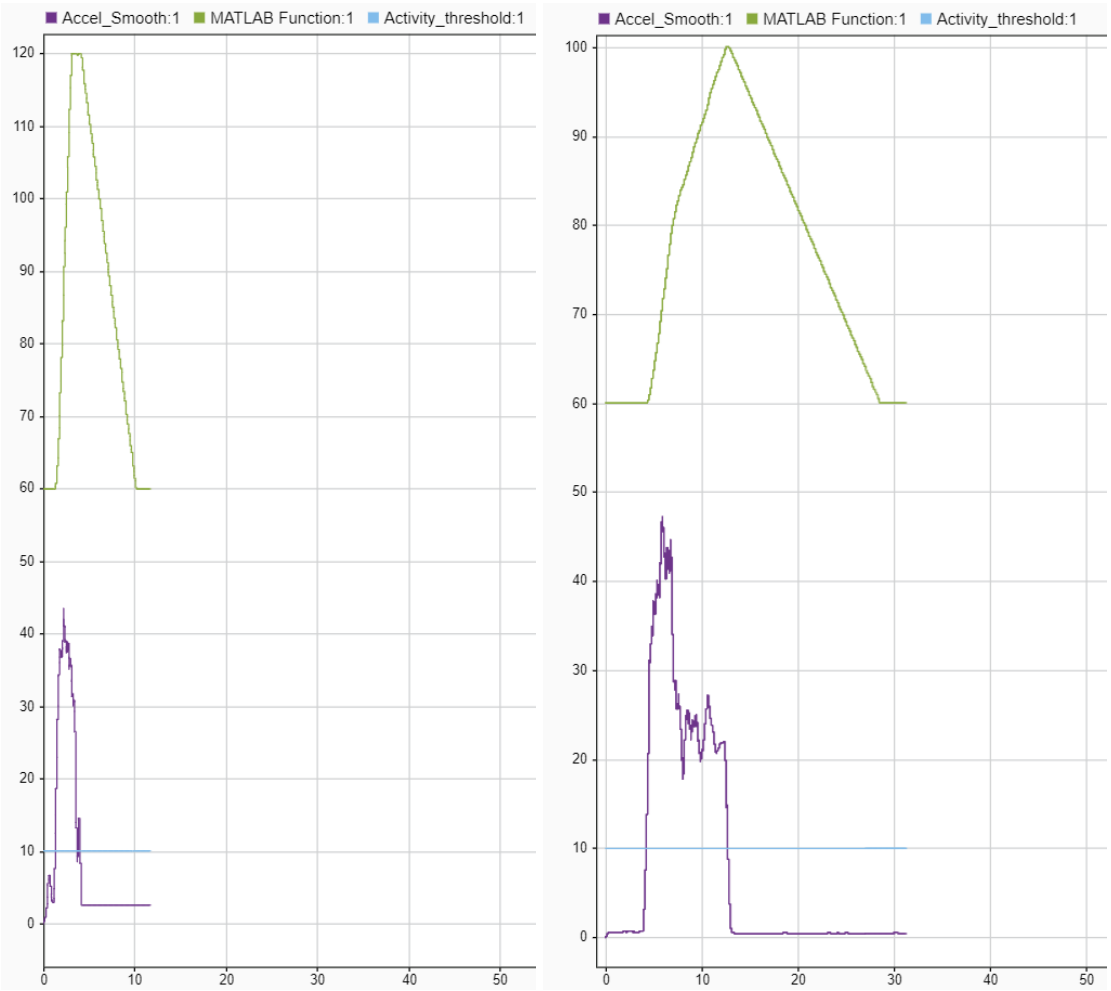


Figure 56. Changing Recovery Time, Target Rate(Green), Activity (Purple)

Recovery Time 1 (Left), Recovery Time 5 (Right)

DCM Cases

The following couple of test cases experiment with the robustness of our login and registering system. Many regulatory standards require robust testing of security systems as well. Since we are creating a medical device, customer information and security is extremely important in order to comply with legal and industry standards, such as HIPAA.

Test Case 1: Invalid Username and Invalid Password

Test Justification and Purpose

The purpose of this test is to determine if the DCM provides the minimum amount of security and rejects users from entering in "garbage data" into the login page to bypass it and access the pacemaker's parameters.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
1	Invalid Username and invalid Password	**Username:** x **Password:** x	Invalid Login Rejection Message	Invalid Login Rejection Message	Pass

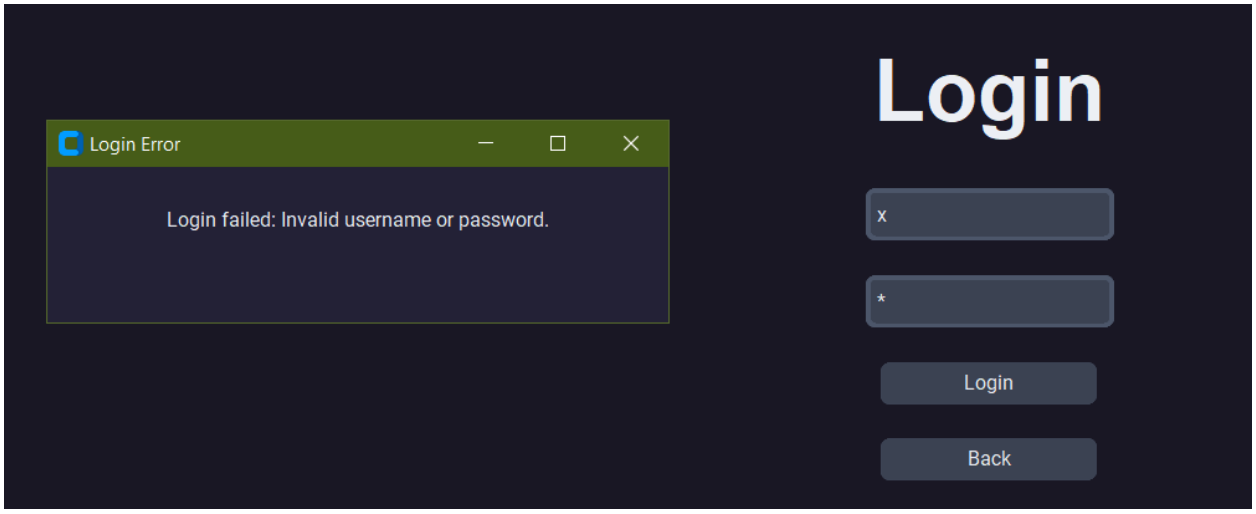


Figure 57. DCM - Login Error Message Screen When Logging in with Invalid Username and Invalid Password.

Test Case 2: Valid Username and Invalid Password

Test Justification and Purpose

The purpose of this test is to determine if the DCM protects the user's parameters and account information if a valid username is entered but the password is incorrect. This would likely be the case if someone was intentionally attempting to hack into a known registered user's account. Additionally, since our DCM encrypts users' passwords but not their usernames, this test would be prevention against cyberattacks following a data leak.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
2	Valid Username and invalid Password	**Username:** 1 **Password:** x	Invalid Login Rejection Message	Invalid Login Rejection Message	Pass

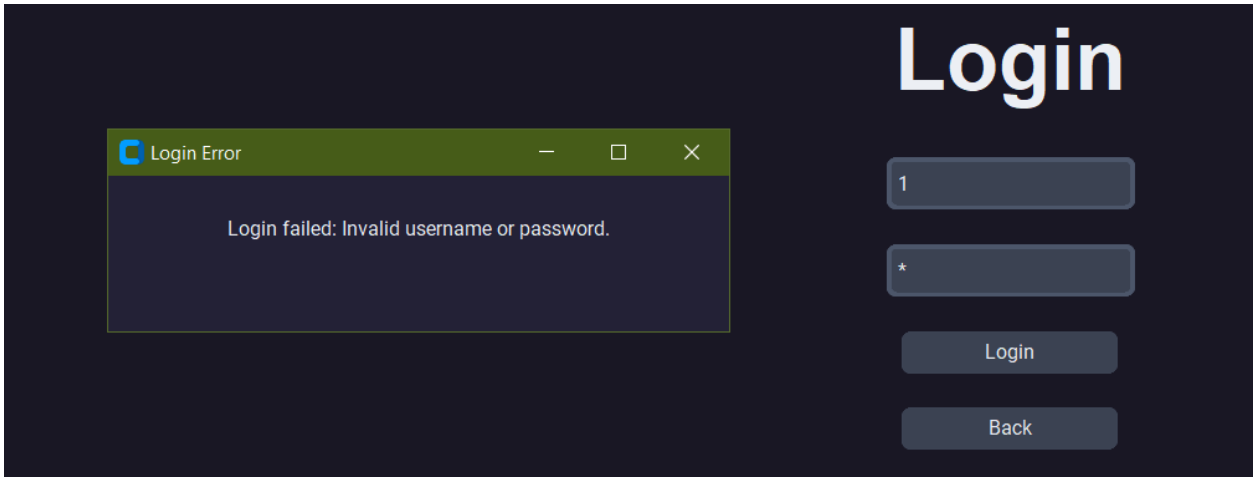


Figure 58. DCM - Login Error Message Screen When Logging in with Valid Username and Invalid Password.

Test Case 3: Invalid Username and Valid Password

Test Justification and Purpose

This test scenario is essential to ensure that the authentication system is secure. It helps verify that unauthorized users cannot access the system by inputting a random or guessed username with a valid password format. This kind of testing helps in preventing brute force attacks where attackers try different username combinations with potentially valid passwords.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
3	Invalid Username and valid Password	**Username:** x **Password:** 1	Invalid Login Rejection Message	Invalid Login Rejection Message	Pass

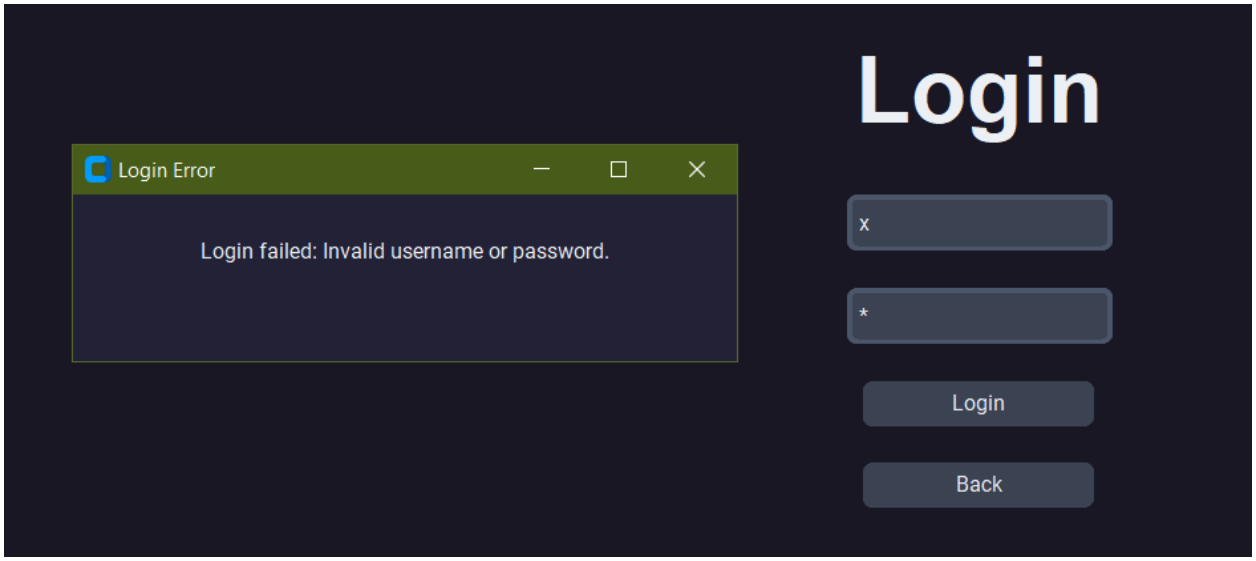


Figure 59. DCM - Login Error Message Screen When Logging in with Invalid Username and Valid Password.

Test Case 4: Attempting to Register More than 10 Users

Test Justification and Purpose

As per our requirements, our pacemaker software is meant to be able to register a fixed number of users, which in our case is a maximum of 10 accounts.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
4	Creation of 10 accounts allotted, attempted registering of an 11th account.	**Username:** 11 **Password:** 11	Invalid Account Creation Due to Maximum Accounts Reached	Invalid Account Creation Due to Maximum Accounts Reached	Pass

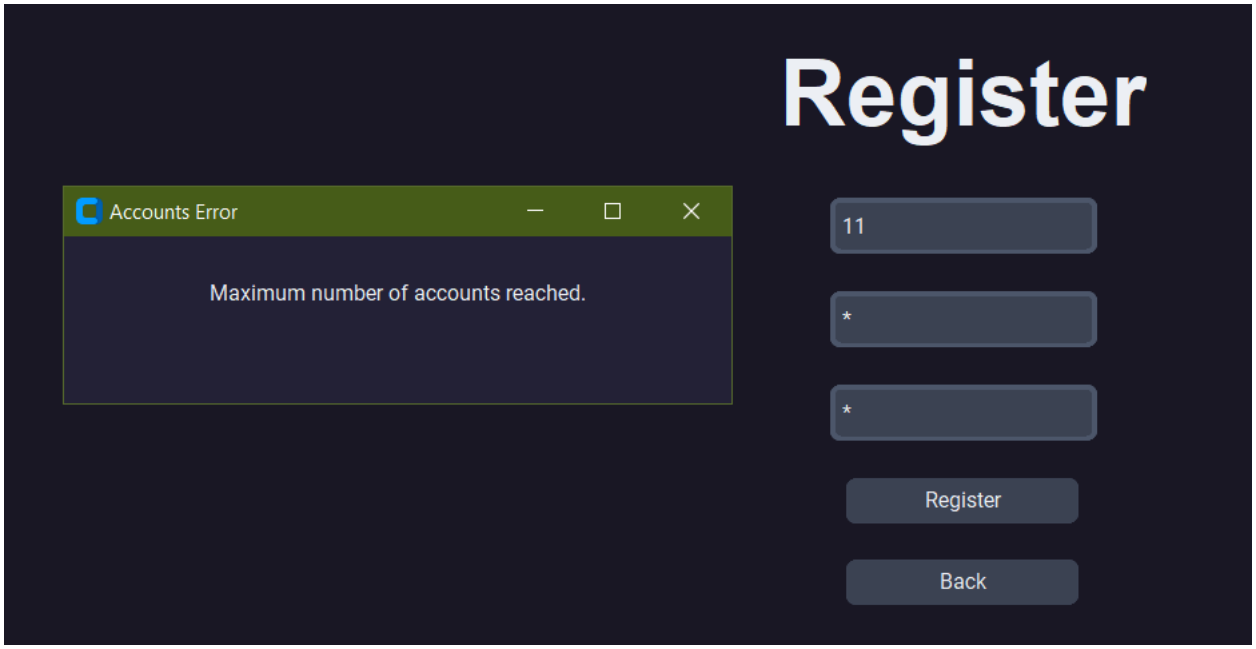


Figure 60. DCM - Account Creation Error Screen When Maximum Number of Accounts is Reached.

Test Case 5: Attempting to Register Without a Password

Test Justification and Purpose

The following test case is done so that a user cannot register with an empty password, preventing a potential data leak if a malicious source correctly guesses the correct username. As per the requirements, users must register with both a valid username and password in order to be eligible to operate the pacemaker.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
5	Attempting to register without a password.	**Username:** 11 **Password:** “”	Invalid Account Creation Due to Empty Password Field	Invalid Account Creation Due to Empty Password Field	Pass

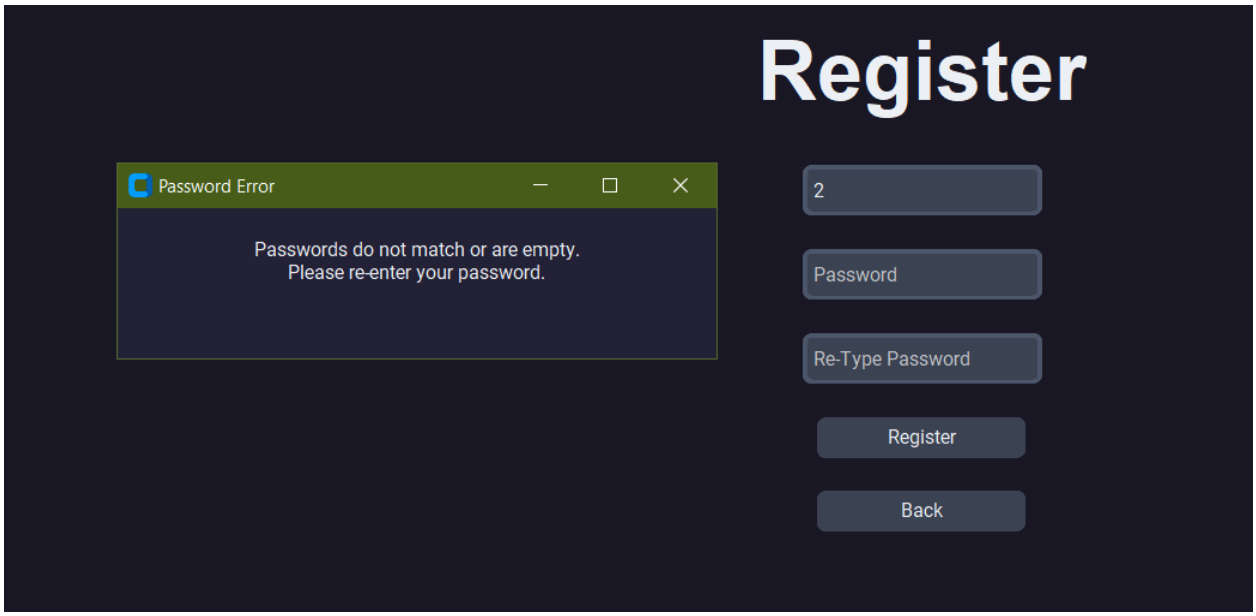


Figure 61. DCM - Password Error Message Screen When Password Field is Empty.

Test Case 6: Password and Re-typed Password Don't Match While Registering

Test Justification and Purpose

This scenario is to ensure that the user is meaningfully entering in their intended password while creating their account. If a typo were to happen when creating their account and entering in their password the first time, the likelihood of that same typo happening when entering in the password for a second time is much less. Rather than having the user create an account with their incorrect password and having to troubleshoot logging into their account, having the user retype their password and ensuring both fields match adds an extra layer of protection.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
6	Registered passwords do not match	**Username:** 1 **Password:** 1 **Re-typed Password:** 11	Invalid Account Creation Due to Mismatched Passwords	Invalid Account Creation Due to Mismatched Passwords	Pass

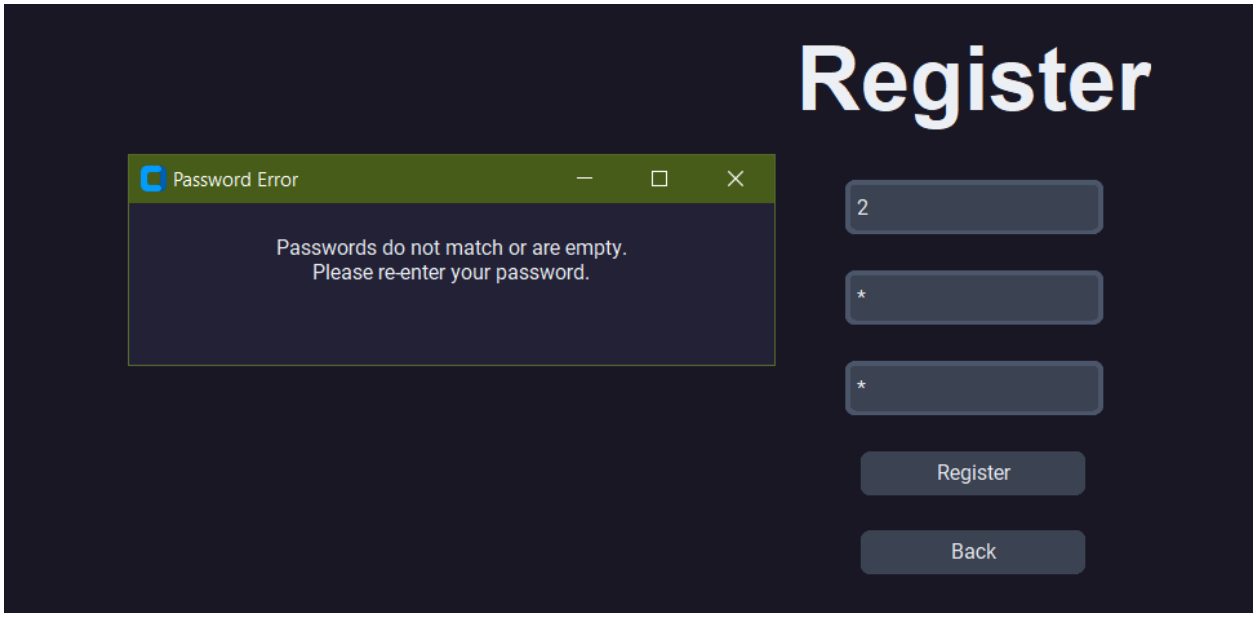


Figure 62. DCM - Password Error Message Screen When Passwords Do Not Match.

Test Case 7: Attempting to make the Upper-rate Limit < Lower-rate Limit

Test Justification and Purpose

In our case, since we are dealing with a pacemaker any misinput could lead to a potential error in the pacemaker function – having a severe impact on the user's risk of having a heart attack. The following test is an edge case, to see if mismatching the two rate limits results in an error.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
7	Upper rate limit is less than lower rate limit	**Lower Rate Limit:** 300 **Upper Rate Limit:** 200	The pacemaker operates normally without any errors	The pacemaker operates normally without any errors	Pass

In our case, when we sent the above packet to the pacemaker. It passes through the DCM without any errors:

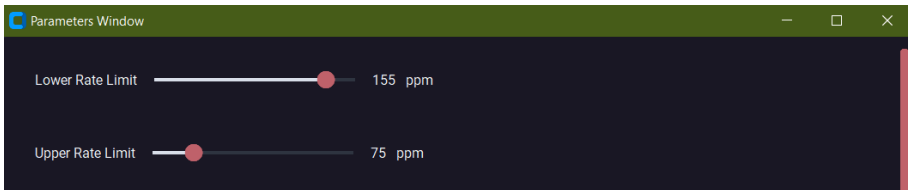


Figure 63. DCM - Parameters Screen With LRL Set to a Higher Value Than URL.

```
values [0, 155, 75, 120, 5, 5, 1, 1, 4, 4, 250, 320, 320, 4, 30, 8]
check [0, 155, 75, 120, 5, 5, 1, 1, 4, 4, 250, 320, 320, 4, 30, 8]
sent packets verified
```

Figure 64. Command Terminal Confirmation That Parameters were Sent Via Serial Communication.

However, in our Simulink logic, we take the lower of the two values between LRL and URL to use for the maximum sensor rate.

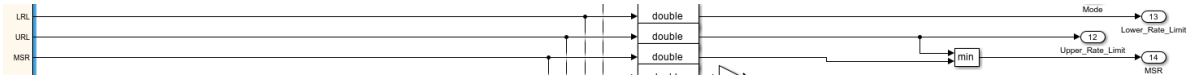


Figure 65. Simulink Logic Showing That the Minimum of the Two Values is Used For LRL and the Other is Used for URL.

Test Case 8: Checking if Parameters are Specific to Each User

Test Justification and Purpose

This test case is to ensure that the parameters of the pacemaker are user-specific. Data-leaking and personalized care for the pacemaker is essential as different patients will have different underlying conditions and specific parameters to them.

The following test case was checked with the parameters set when logged-in and then double-checked upon a second login attempt:

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
8	Parameters are unique to each logged-in user	**User 1→LRL:** 60 **User 2→LRL:** 110	**Upon Re-login:** **User 1→LRL:** 60 **User 2→LRL:** 110	**Upon Re-login:** **User 1→LRL:** 60 **User 2→LRL:** 110	Pass

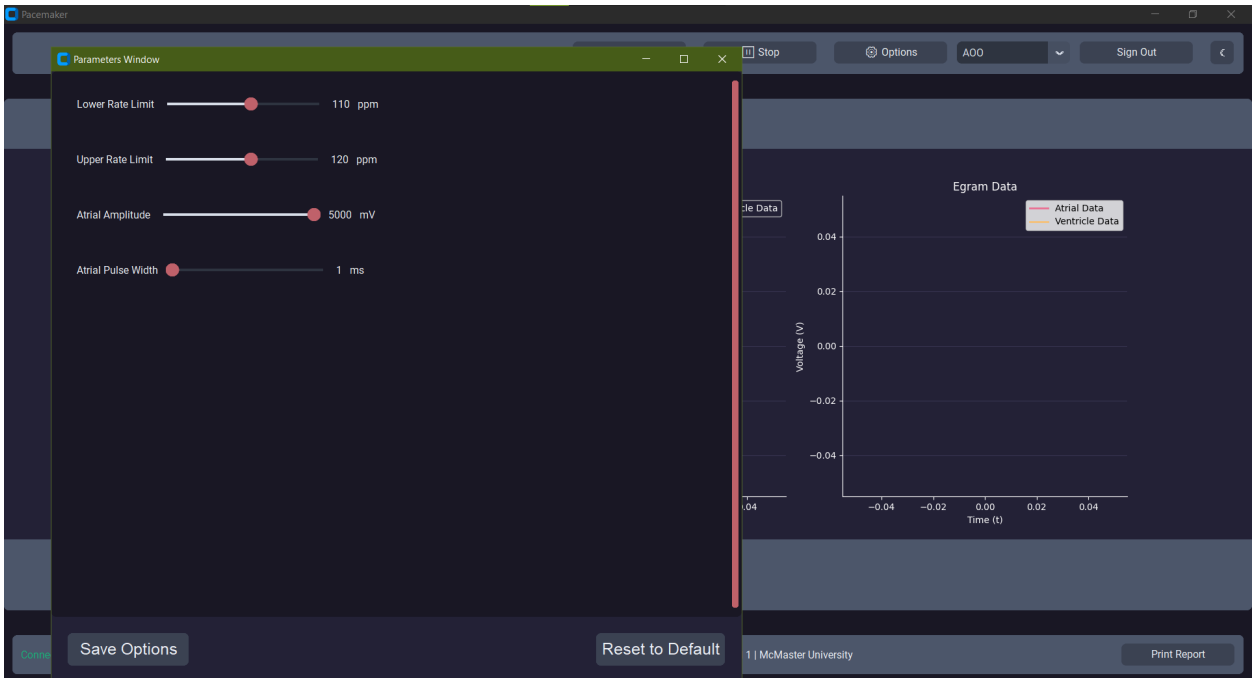


Figure 66. DCM - Parameters Screen For User 1 with LRL set to 110.

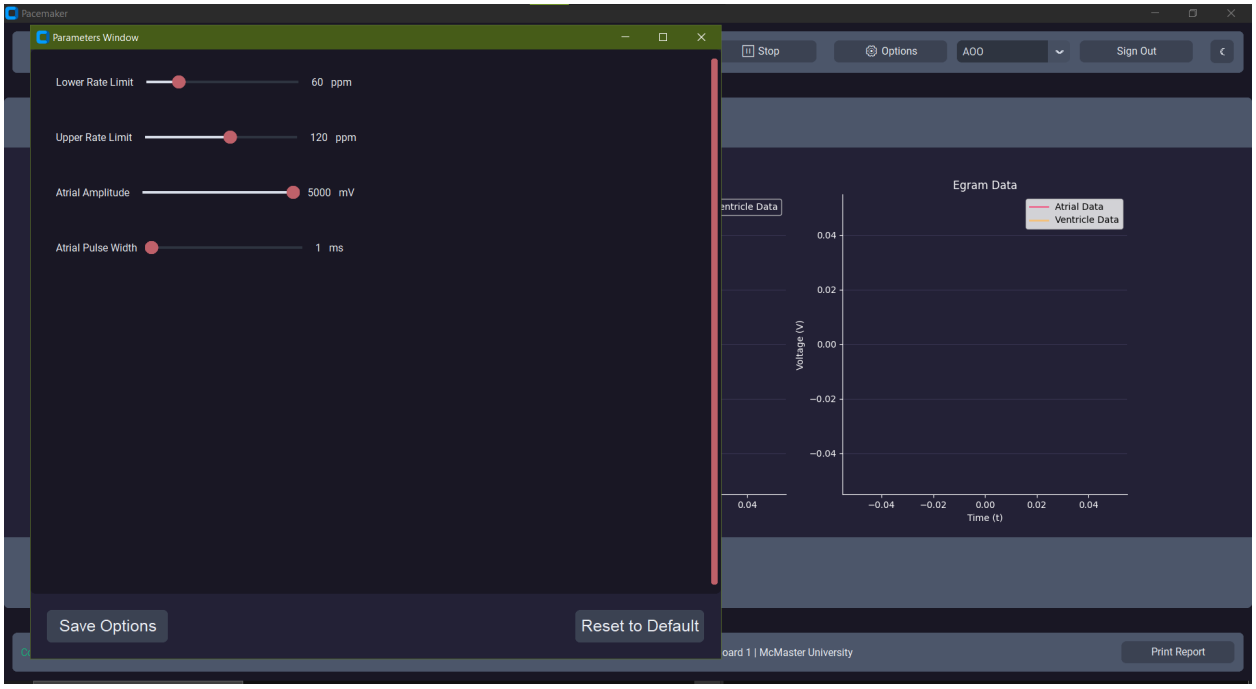


Figure 67. DCM - Parameters Screen For User 2 with LRL set to 60.

Figure 68. JSON Saved Parameters File Confirming That the Parameters were Uniquely Saved for User 1.

Figure 69. JSON Saved Parameters File Confirming That the Parameters were Uniquely Saved for User 2.

All test conditions are run with nominal input parameters, 60bpm heart rate, and 30ms AV delay.

Test Case 9: The admin is able to delete non-admin accounts

Test Justification and Purpose

In our case, since we are dealing with a pacemaker any misinput could lead to a potential error in the pacemaker function – having a severe impact on the user's risk of having a heart attack. The following test is an edge case, to see if mismatching the two rate limits results in an error.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
9	Admin account can delete non-admin accounts	Deleting User 4	User 4 should disappear from the JSON Users File	User 4 disappeared from the JSON Users File	Pass

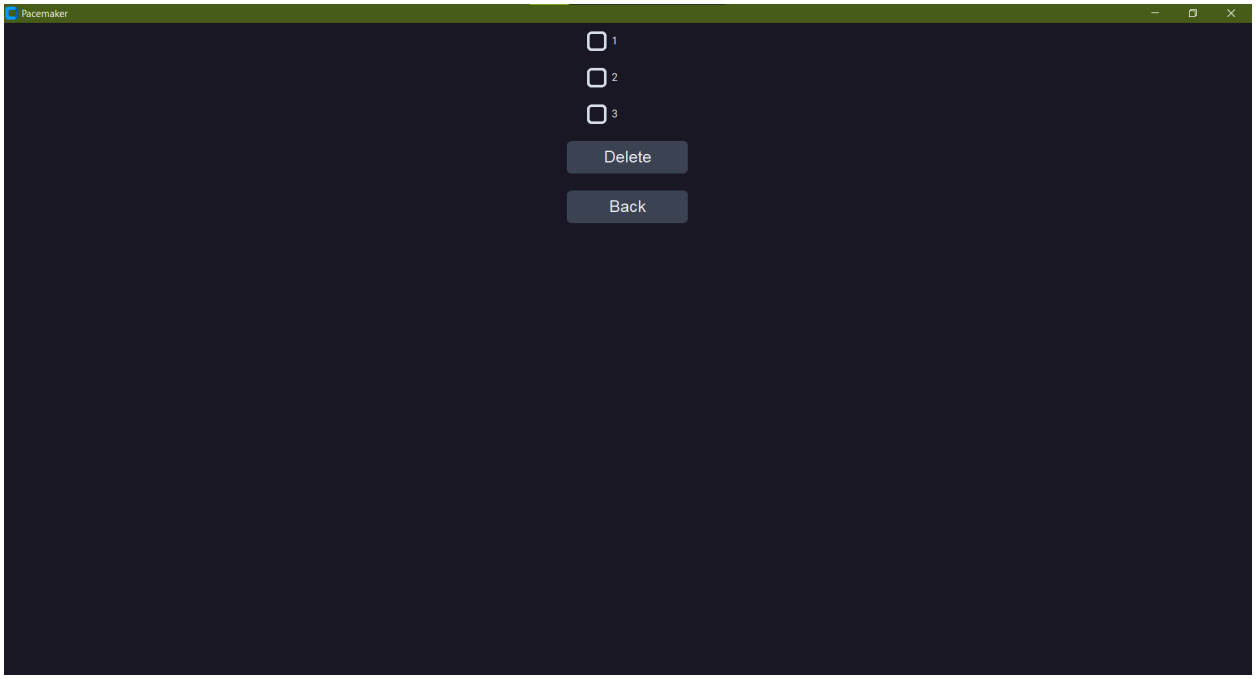


Figure 70. DCM - Admin Account Managing Screen After User 4 Was Deleted.

```
{
  "Username": "4",
  "Password": "d4735e3a265e16eee03f59718b9b5d03019c07d8b6c51f90da3a666eec13ab35",
  "Saved Parameters": {
    "AOO": {
      "Lower Rate Limit": 60.0,
      "Upper Rate Limit": 120,
      "Atrial Amplitude": 5000,
      "Atrial Pulse Width": 1
    },
    "AAI": {},
    "VOO": {},
    "VVI": {},
    "AOOR": {},
    "AAIR": {},
    "VOOR": {},
    "VVIR": {}
  }
}
```


Figure 71. JSON File Before Deleting User 4 Account.

Figure 72. JSON File After Deleting User 4 Account Showing User 4 is Gone.

Test Case 10: The GUI Does Not Allow for Incorrect Input Data

Test Justification and Purpose

This test case ensures that the DCM GUI does not allow the user or anyone who should be operating the pacemaker to enter incorrect or non-allowable input data into the pacemaker parameters. Since computers operate any instructions given to them, if a fatal value such as a negative or high integer were inputted, then the patient's safety could be at risk.

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
10	Move the slider to the extreme ends and attempt to enter in values.	Moving LRL slider to the maximum and minimum locations	The slider should lock and prevent unspecified values from being inputted	The slider locks and prevents unspecified values from being inputted	Pass

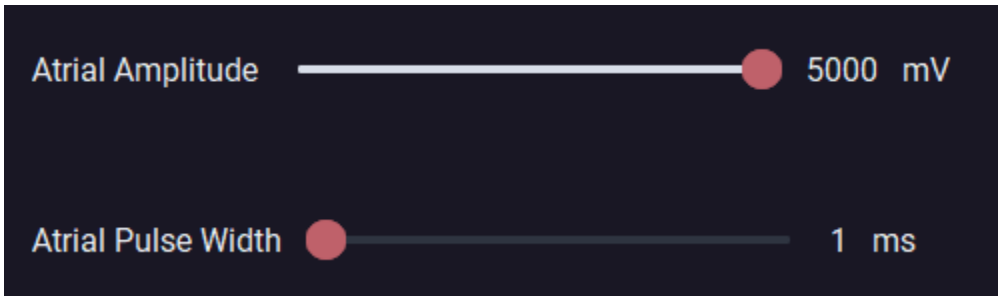


Figure X. DCM - Parameters Screen With Sliders at the Maximum and Minimum Values Demonstrating the Values lock at the Extremes.

Serial Test Cases

The following are test cases to verify if the serial communication that we implemented has all of the functions that we expect it to perform:

ID	Test Case	Input	Expected Result	Actual Result	Pass/Fail
1	Testing if the serial ports selected is correct for each board	Connected the board to the DCM	Port = “COM5” or “COM3” for each board	Port = “COM5” or “COM3” for each board	Pass
2	Testing if the sending packets are functioning	Sent default parameters using the send() function	In terminal, print(“sent”)	“sent” in terminal	Pass
3	Testing if the receiving packets are functions	After sending a packet, receiveSerial() should return a list of values on the pacemaker	List of values that were sent including the mode	[0, 155, 75, 120, 5, 5, 1, 1, 4, 4, 250, 320, 320, 4, 30, 8]	Pass
4	Testing if the egram function is pulling data	Once connected to the board, egramPull() should return a set of two doubles	When atr and vent signal are off in heartview it should return two constant doubles	[0.49878689646720886 , 0.49944305419921875]	Pass
5	Testing if the egram pull rate is adequate for creating a correct graph	Once connected to the board and start() is called, the graph shape should be somewhat correct [Minimum pull rate: 0.1 secs]	Graph should output and be clear	Graph outputs and is clear, maxima and minima are distinct	Pass
6	Testing if packets sent are the same as the packets received	When “save options are clicked” the function send_Data_checked() should verify if the sent and received are the same	Sent data and data received immediately after sending should be the same	Sent data and received data are the same	Pass

8	Testing if the units sent are in the range of everything	Attempt to set a value outside of the accepted range of the values for each parameter	The user should not be allowed to input a value out of range	The user is not allowed to input a value out of the range of the parameter	Pass

Problems and Challenges

Pacemaker

Assignment One Challenges

One of the biggest challenges faced over the course of assignment one had to do with testing of stateflows and getting heartview to function properly. The majority of the initial testing was conducted on only one of the assigned boards, however it was failing to display any paced signals in any mode. It was later determined that communication between the pacemaker and heart on that particular board was faulty and that no signals were getting through. We were able to get our Simulink running on our other assigned board however it was only after quite a lot of wasted time working with a faulty board.

Another challenge faced was with sensing for AAI and VVI. Getting feedback from heartview was a struggle and after many hours of debugging it was determined a variable known as Front end control was missed. In the circuity this acts as the switch to enable the entire sensing circuity and was the reason why no inputs we being received.

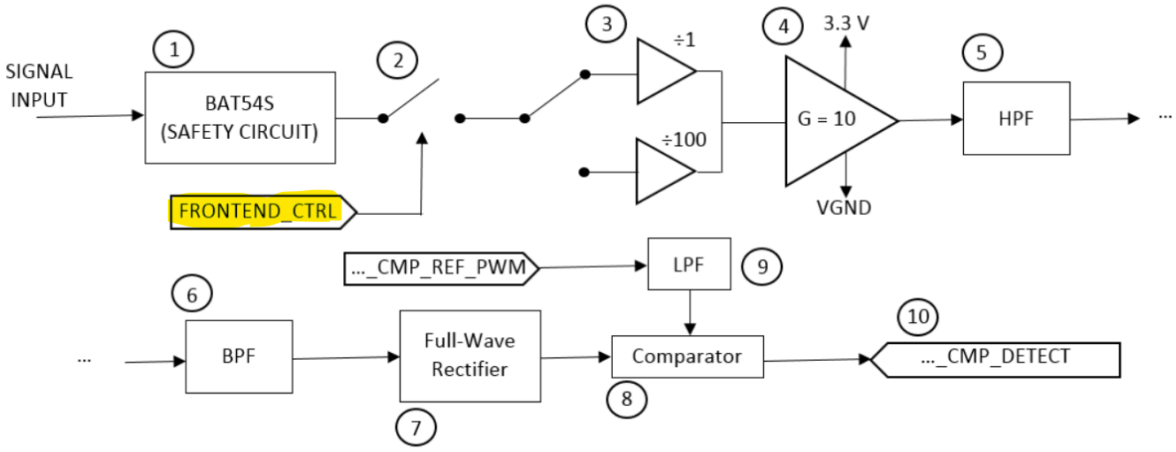


Figure – Frontend Control

Assignment Two Challenges

Our first priority after our assignment one demo was to correct any functionality that had been missed or implemented incorrectly. For us this had to do with the AAI and VVI modes. Our initial interpretation of these modes was for the pacemaker to constantly be sensing for a natural heart rhythm, and if no rhythm was present to then send artificial paces. In an example where the natural heart is set to 60 and the LRL is set to 120, the pacemaker stayed off until the natural heart was turned off. After our demo, we learned that what was actually required was for our pacemaker to maintain the LRL irrespective of if the natural heart was on or off. This meant that in the same example with natural heart at 60 and LRL at 120, we should see pacemaker paces in between every natural heart pace so as to maintain the 120 LRL. We corrected this issue by adding a waiting and sense state into the stateflow and altering the timings of the standby periods between them.

Another issue we faced early on had to do with Simulink's external mode, and the monitor and tune function in the hardware tab. Being able to actively monitor signal changes in real time was critical for ease of development and testing of our rate adaptive implementation. That being said, monitor and tune requires a feature called external mode (which establishes a communication channel between Simulink and the target hardware) to be on. Unfortunately, when it came to implementing serial communication, external mode cannot be on as the serial port would already be occupied by the DCM's connection. We encountered an ambiguous error related to this issue for a long time without any idea of how to fix it, and when we finally discovered it meant external mode needed to be disabled, we were unable to turn it off. In the end we had to copy all of our work into a new simulink file with the feature turned off.

Staying on the topic of rate adaptive pacing, another issue we discovered while tuning and troubleshooting had to do with the raw accelerometer data. As previously mentionned in rate adaptive design, we learned that the accelerometer inputs 1g at baseline, as this is the force of gravity pushing down on us. Before we learned this we had trouble experimentally determining an activity threshold value and had to vigorously shake the board to see any noticeable change. Once we discovered the constant 1g os acceleration, we simply subtracted 1 from the accelerometer's z input.

DCM

Throughout the entirety of the design process for the DCM, we faced a variety of challenges that presented as follows:

Assignment One Challenges:

1. The creation of our parameter window was challenging at the beginning. As we aimed to not hardcode each of the text, sliders, and slider captions with the end goal of creating cleaner code. The main issue that we had was accurately formatting and getting each of the elements to appear without any issues.
2. Due to the separation of classes between the main and parameter window, we found a lot of issues inserting variables into the arguments of the class declarations. This was mainly due to errors involving customtkinter and prevented us from connecting the state variable from the App() class to the ParameterWindow() class.
3. While implementing serial communication between the DCM and Simulink, there were a number of parameter difficulties that we encountered including finding the appropriate sensitivity that would produce the desired responses in Heartview
4. Unit conversion between the simulink and the DCM was also a temporary challenge that needed to be accounted for, as sometimes the magnitude of units would be altered in the DCM to remove decimal values, but needed to be adjusted to reflect the values sent by the simulink.
5. Something that propagated throughout the entire project was difficulties with our GitHub branches, mainly due to merge conflicts created by (at the time) poor programming practices.

Assignment Two Challenges:

In assignment two, we faced a variety of issues that we had to fix – with most of these issues involving integration:

1. From assignment one, we had to fix a variety of errors that were present in our code, the first was that when a user selected a mode the parameters they could edit would not simply be the ones that they needed to edit. To fix our mode selection code, we had to redo our implementation of user data from .txt files to .json files. Doing this required a lot of refactoring of our code to accept dictionaries as inputs.
2. After refactoring our code to accept .json files, we faced an issue where we could not get the parameter window to update when selecting a mode, saving options, and resetting options. This error was due to using an iterative loop to generate our parameters window through the InputFrame class - since our previous code was hardcoded to text files, we had to change it to work properly.
3. We couldn't do work unless a member of the DCM had a board and we only had two boards.
4. We had to implement serial communication which posed a variety of issues due to linking items from the GUI to Simulink
    a. Our serial did not work until we fixed our units
    b. A variety of our code logic for sending and receiving data in the GUI did not work for different modes and required a rewrite
5. We had to make our DCM work without the board being plugged into the device and be able to be hot swappable, to do this we had to make our device work realtime:
    a. To implement this feature we tried to use multithreading but it did not function as intended and slow down our code by a variety of factors
    b. Instead we implemented the .after() function
6. We had to make our DCM display the graph real time, to do this we used a class for a subframe that updated using .after()
7. We spent a great deal of time trying to make tkinter look good through images, which resulted in a bunch of image files that required directories, however since the group members on DCM and on GUI were using mac and windows we had to use python os to create directly
8. Serial ids on mac were different than windows and would not work properly
9. Throughout our code we had a variety of merge problems that impeded our workflow

Future Project Prospects

No matter how far innovation and creativity takes us, there is always room to grow and evolve given time and newly emerging fields. The following sections describe additional features we would incorporate if we were to take this project further in the future.

Pacemaker

Implementation of All other pacing modes

- Objective: Implement all other required pacing modes as specified in the Pacmaker system specification document

- **Rationale:** In order to provide comprehensive bradycardia therapy having all of the possible pacing modes is critical.

- **Examples:** AAT, VVT, VDD, DOO, DDI, DDD, VDDR, DOOR, DDIR, DDDR

Implementation of all other pacing parameters

- Objective: In addition to the new pacing modes, certain modes have specific parameters which need implementation.

- **Rationale:** For correct functionality, new modes require their specific parameters

- **Examples:** Dynamic AVdelay, sensed AVdelay offset, PVARP extension, ATR fallback mode, ATR fallback time,

Implementation of rate smoothing an hysteresis

- Objective: Add rate smoothing and hysteresis to the modes that require them as per the Pacemaker system specification document

- Rationale: Rate smoothing will prevent precipitous changes in the rate due to rapid or sudden blips in activity, and hysteresis encourages self pacing more during exercise by waiting slightly longer after senses.

- Examples: AAI, VVI, DDD and their rate adaptive equivalents

DCM

Enhanced Data Security and Privacy Protocols
- **Objective:** Implementing advanced security measures to protect patient data.
- **Rationale:** As technology evolves, so do the threats to data security. Implementing robust encryption and security protocols will safeguard against unauthorized access and ensure HIPAA compliance.
- **Examples:** Two factor authentication, email verification, SSN/SIN verification, Health Card Verification.

Further Analytical Functionality
- **Objective:** Implementing functionality and analytical measures to analyze patient data.
- **Rationale:** Currently our pacemaker simply outputs the current data that from the pacemaker, to be a viable market pacemaker software it should be able to run calculations and use patient data to diagnose efficacy of the software.
- **Examples:** Errors, warning signs, report data, calculations using graphs created, etc.

Cloud Integration and Remote Monitoring Capabilities
- **Objective:** Enable cloud-based data storage and remote monitoring.
- **Rationale:** Integrating cloud storage will allow for real-time data analysis and remote monitoring, enhancing patient care through timely interventions. This also opens avenues for AI-driven predictive analytics. Local storage should still be used as a backup.
- **Examples:** Having a secure cloud storage system separate from existing cloud storages solely for patient data.

User Interface Personalization and Accessibility Features
- **Objective:** Develop a more intuitive and accessible user interface.
- **Rationale:** An adaptable UI that caters to various user needs, including those with disabilities, will enhance usability and ensure the DCM is more patient-friendly.
- **Examples:** Although our accessibility is quite strong, we can always integrate dynamic features that appeal to more patient groups such as visually impaired individuals. Perhaps auditory cues can be an additional feature.

Multi-Language Support
- **Objective:** Include multiple language options for the DCM interface.
- **Rationale:** To cater to a diverse patient population, adding multi-language support will make the device more accessible to non-English speaking users.
- **Examples:** Our pacemaker was only implemented in English, but we can use an API for multi-language-report for the parameters and login prompts.

Integration with Wearable Health Devices
- **Objective:** Sync the DCM with other health monitoring wearables.
- **Rationale:** This integration will allow for a more comprehensive health monitoring system, providing detailed insights into the patient's overall health status.
- **Examples:** Integrating with Apple watches, Fitbits, Whoops, Samsung wears etc.

Development of a Mobile App Companion
- **Objective:** Create a mobile application for the DCM.
- **Rationale:** A mobile app will offer convenience, allowing patients and healthcare providers to access data and receive notifications on-the-go.
- **Examples:** Having a mobile app of some kind to integrate with health technology is the main standard and most convenient way to lower the barrier of patients being able to access their health information. Companies such as Apple, Whoop, Fitbit etc. all have apps that integrate with their health technology.

Machine Learning for Predictive Analysis
- **Objective:** Incorporate machine learning algorithms for predictive health analysis.
- **Rationale:** Utilizing machine learning can help predict potential cardiac events and improve preemptive care strategies.
- **Examples:** Patients can be told which lifestyle changes they can implement for better cardiac health.

Expansion of Data Management Capabilities
- **Objective:** Enhance the data management system to handle more complex datasets.
- **Rationale:** As medical data becomes more intricate, a sophisticated data management system will be necessary to process and store this information efficiently.
- **Examples:** Expanding to a cloud-storage or hardware server to handle larger amounts of patient data.

Interoperability with Hospital Information Systems
- **Objective:** Ensure DCM compatibility with various hospital information systems.
- **Rationale:** Interoperability with hospital systems will streamline patient care, making it easier to integrate pacemaker data into existing medical records.
- **Examples:** If a patient goes into cardiac arrest or has a pacemaker malfunction a healthcare worker can offer support or notify the authorities that there is a risk to the patient’s life.

Continuous Software Updates and Maintenance
- **Objective:** Establish a protocol for regular software updates and maintenance.
- **Rationale:** Regular updates will ensure the DCM stays current with the latest technological advancements and security patches, maintaining its reliability and effectiveness.
- **Examples:** Updating with the newest real-time updating software to optimize signal speed.

Conclusion

As we bring this comprehensive report on our pacemaker project to a close, it's imperative to reflect on the multifaceted journey we have undertaken. This journey is marked by a seamless blend of cutting-edge engineering and a deep commitment to patient-oriented design. Our venture has transcended the traditional application of mechatronics and embedded systems in healthcare, showcasing the transformative potential of integrating sophisticated technologies such as MATLAB Simulink and Heartview. These integrations have been crucial in crafting a precise and dependable pacemaker architecture. A notable milestone in our project was the development of the Device Controller Monitor (DCM), which stands out for its intuitive interface, enhanced multi-user functionality, and robust data security, thereby setting new benchmarks in medical device technology.

Throughout this endeavor, we faced numerous challenges, ranging from technical hurdles in developing the DCM and Simulink models to the intricacies of software and hardware integration. These challenges served as invaluable learning experiences, significantly honing our problem-solving skills and enriching our understanding of the complexities inherent in medical device development.

Looking to the future, the possibilities for enhancing both the DCM and pacemaker technology are vast. We envision advancements such as sophisticated data security protocols, seamless cloud integration, customizable user interfaces, and full integration with hospital systems. Additionally, the potential integration of machine learning for predictive analytics, alongside the development of a mobile application companion, reflects our commitment to continuous technological evolution. In the realm of Simulink development, the incorporation of additional states and pacing requirements is critical to creating a comprehensive solution capable of addressing all bradycardia conditions and enhancing patient comfort.

In summary, this project represents a significant landmark in the field of cardiac pacemaker technology, embodying the essence of collaborative innovation and patient-centric design in healthcare technology. As we conclude this chapter, we eagerly anticipate the ongoing evolution and impact of our work, fueled by our unwavering dedication to improving patient care and advancing the frontiers of medical technology.