MECHTRON 3K04: Software Development

Assignment 2

Group 35

Rebecca Penny, 400440507

Carter Sankoff, 400476177

Simrin Leonard, 400444881

Braeden Marchant, 400441243

Alex Walmsley, 400475004

Teaken Brazeau, 400463576

Contents

Table of Figures

Part 1

Requirements and Design Decisions

Simulink Requirements:

Pacing Modes

The are many modes and methods of pacing and sensing that the pacemaker can provide but for this particular assignment the group is only interested in programing single chamber pacing adhering to NBG codes AOO, AAI, VOO, and VVI. More detail on the variables relevant to these modes is as follows.

Position:	Letters Used and Meaning:
1	The first position gives the chamber(s) that are being paced by the pacemaker. A – Pacing of the atrium V – Pacing of the Ventricle O – No pacing D – Dual
2	The second position represents the chamber(s) being sensed. A – Sensing of the atrium V – Sensing of the Ventricle O – No sensing D – Dual
3	The third position represents the desired response to sensing. O – None T – Triggered I – Inhibited
4	The fourth position represents the programmability or rate modulation R – Rate Modulation

Given this, the overall requirement from the Simulink code was to program the pacemaker to pace, sense, and inhibit either the ventricle or atrium of the heart based on a selection from the user. For assignment two, addition modes and variables become relevant, and the Simulink is modified to include NBG modes AOOR, VOOR, AAIR, VVIR, and DDIR. The meaning of these codes is specified in the above table.

Software Requirements

In order to effectively program the necessary four pacing modes, the program must consider and control several parameters as they are received. Described below are all the necessary programmable parameters for the software to function effectively.

Mode:

The desired pacing mode (of those described in the previous section) for the pacemaker to operate in can be selected.

Lower Rate Limit (LRL):

The number of generator pace pulses delivered without sensed intrinsic activity or Sensor controlled pacing.

Upper Rate Limit (URL):

The maximum heart rate that the pacemaker tells the heart to achieve.

Pulse Amplitude (Atrial and Ventricle) Regulated:

The amplitude of pulse sent to the heart when pacing in either atrial or ventricle modes.

Pulse Width (Atrial or Ventricle):

The width of the pulse sent to the heart when pacing in either atrial or ventricle modes.

Sensitivity (Atrial or Ventricle):

The ability of the device to sense the hearts activity. Necessary to determine how the pacemaker is to react with the heart.

Refractory Period (Atrial or Ventricle):

The period in which the pacemaker ignores any sensed input after a pacing event.

Heart Input Signal (Atrial or Ventricle):

Necessary to the sensing circuitry, this will return high when the voltage is over the set threshold voltage and low otherwise.

Maximum Sensor Rate:

The maximum pacing rate allowed as a response to accelerometer data.

Recovery Time:

The time required for the rate to fall from the maximum sensor rate to the lower rate limit after activity below the threshold is detected.

Reaction Time:

Similarly to the recovery time, the time decided upon for the rate to climb from the LRL to the MSR when driven by activity.

Response Factor:

Determines at which increment the sensor rate will change in response to activity.

Serial Communication:

For the DCM and the Pacemaker/Simulink to function wholistically and correctly, serial communication module must be worked in to the program design.

Simulink Design Decisions:

For assignment one, the Simulink design was intended to be as modular as possible. It can be switched between four modes (AOO, VOO, AAI, VVI) which will run the same modules but with different behaviours. Given this intent the code separates into a block containing and converting all necessary inputs from external sources (the board) in order to be passed module that will conduct the pacemaker functions. This input block was designed to be internally contained and convenient to make changes to the more controllable parameters. Once into the pacemaker function the module will behave differently based on the selected mode.

Initial:

VOO

As this mode does no sensing or inhibiting, it is most simply a cycle of charging the primary on board capacitor, directing voltage to pace the ventricle, and then discharging the capacitor before restarting the cycle.

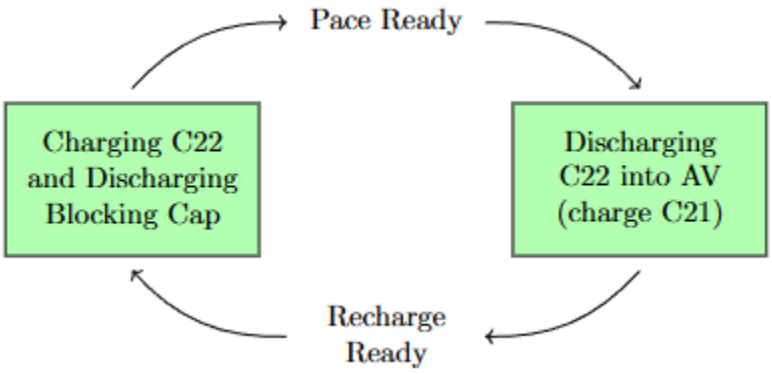


Figure 1: Pacing cycle flowchart.

Much of the electrical process behind this cycle is condensed nicely in hardware form on the pacemaker and the program is essentially there to accept inputs and direct outputs to and from the desired pins of the pacemaker board. One value of particular interest for the design is the period of this cycle. Utilizing the values of the lower rate limit, the ventricle pulse width, and the refractory period, and ideal cycle can be determined within the program based on parameters given in the beginning.

VVI

When set to this mode, there is additional requirements outside of the charging, pacing, and discharging state. The program must utilize a sensing function in order to ensure pacing is only done when necessary (there is no or limited signal from the heart).

In this mode the program does all the same functions as VOO but does not pace the ventricle unless no signal from the heart is detected. If the input signal sent from the heart is set to high then the program is set to continue charging for a desired period (refractory period) until it will once again sense for a pulse.

If the value returns low, then the sensing function will call on the pacing as demonstrated in VOO until the input from the heart once again returns as high and the program returns to the sensing only state.

AOO

If the mode is set to AOO, all the same functions as VOO apply but pacing is directed to the atrium instead of the ventricle. All variables containing “VENT…” will be run through those containing “ATR…” instead.

AAI

If the mode is set to AAI, all the same functions as VVI apply but pacing and sensing are directed to and from the atrium instead of the ventricle. All variables containing “VENT…” will be run through those containing “ATR…” instead.

Rate Adaptive Pacing:

For assignment two, the modular intention of the Simulink design was continued and as new blocks/modules were added. With the addition of new modes, rate smoothing, and new internal logic the design was kept true to the original and organized to be modular and reduce coupling. For the newly applied rate adaptive modes the functionality is very similar apart from the stages in between the inputs being passed and the modes being called upon.

Although the modes programmed for assignment two are similar in logic to those previously detailed, the key difference in the integration of “Rate Modulation” and rate adaptive pacing modes. Using an onboard accelerometer the new modes (VOOR, VVIR, AOOR, AVIR) must consider the movement of the board and pace the heart accordingly.

Through a series of steps, the accelerometer data is considered and drives an effect on the set pacing of the pacemaker. Firstly, the norm of the accelerometer data is found using the average value of the x, y, and z displacements. Next, the desired rate is calculated by comparing the accelerometer data to a variable referred to as the “activity_threshold”, if the accelerometer data is greater than the threshold, the desired pace is stepped up. If it is lower than the threshold then the rate is gradually stepped down to the LRL.

The program then applies rate smoothing and assigns a new desired rate to ensure the increase of the rate variable isn’t too extreme. The specific details for each rate adaptive mode are as follows.

VOOR:

If the mode is set to VOOR, the program functions very similarly to VOO. The key difference is that instead of continually cycling through charging and discharging as pictured above, the rate for this mode is determined using the rate adaptive rate logic as described above and demonstrated in the Simulink model section.

VVIR:

Similarly, if the mode is set to VVIR, the program functions very similar to VVI as described above. Once again, instead of moving directly from discharging to charging based on a constant cycle, the rate is determined by passing values through rate adaptive logic and adjusting for all necessary values.

AOOR:

If the mode is set to AOOR, all the same functions as VOOR apply but pacing and sensing are directed to and from the atrium instead of the ventricle. All variables containing “VENT…” will be run through those containing “ATR…” instead.

AAIR:

If the mode is set to AAIR, all the same functions as VVIR apply but pacing and sensing are directed to and from the atrium instead of the ventricle. All variables containing “VENT…” will be run through those containing “ATR…” instead.

Dual Chamber Modes:

In addition to the rate adaptive, single chamber modes programmed as a part of assignment two, the group also began to implement dual chamber modes following the module designs and low coupled integration as used before.

The key change with the dual chamber mode is that both chambers cannot be paced simultaneously. An AV delay was integrated to allow for each chamber of the heart to be paced on intervals. Another step taken with the dual chamber modes was adjusting the program to allow for an onboard button press to partially inhibit the pacing of the ventricle.

DOOR:

This mode functions as a alternating call on AOOR and VOOR, the rate adaptive logic is called on similarly and each cycle is alternated between atrial functions and ventricle functions

DCM Requirements (Assignment 1)

Welcome Screen

The interface for the GUI had to include a welcome screen where users can login or register a new username and password. A limit of 10 users that can be allowed to be stored locally also must be implemented.

User Interface

The DCM user interface must be capable of utilizing and managing windows that display text and graphics as well as it should be able to process user positioning and input buttons. It should also display all the input parameters and allow users to review and modify when needed. Additionally, the interface should be capable of recognizing when the DCM and the pacemaker are communicating and visually display this. And finally, it should indicate when a different pacemaker is connected than what was previously.

Present All Pacing Modes

The interface is required to display all applicable pacing modes for the user to select (AOO, VOO, VVI, AAI).

Store Programmable Data

The data users input for the desired parameters should be stored, and when data is being submitted the inputs should be checked and validated. The interface should be able to be verify the data is being stored correctly on the Pacemaker device. The parameters are specified to be as listed in the following table.

Table 1 Initial Programmable Parameters Requirements

Parameter	**Programmable Values**	**Increment**	**Nominal**
Lower Rate Limit	30-50 ppm 50 – 90 ppm 90-175 ppm	5 ppm 1 ppm 5 ppm	60 ppm
Upper Rate Limit	50-175 ppm	5 ppm	120 ppm
Maximum Sensor Rate	50-175 ppm	5 ppm	120 ppm
A or V Pulse Amplitude Regulated	Off, 0.5-5.0V	0.1V	5V
A or V Pulse Width	1-30 ms	1 ms	1 ms
A or V Sensitivity	0-5V	0.1V	N/A
PVARP	150-500 ms	10 ms	250 ms
Ventricular Refractory Period	150-500 ms	10 ms	320 ms

Atrial Refractory Period	150-500 ms	10 ms	250 ms

Data Structures for Egram Data

For future assignments, appropriate data structures should be developed for handling incoming egram data received from the pacemaker. Additionally, this data should be able to be displayed for the users when needed.

Additional Requirements:

Accessibility features should be implemented to make the experience using this interface as easy and enjoyable as possible.

DCM Requirements (Assignment 2)

Expanded Modes and Parameters

The DCM should be updated to include the rate adaptive modes AOOR, AAIR, VOOR, and VVIR. In conjunction with this, the necessary programmable parameters should also be added.

Serial Communication

Serial communication should be implemented to transmit and receive data between the DCM and the pacemaker.

Programmable Parameters

Update the DCM to set, store, and transmit all the parameters including those for the newly added modes. The new parameters and updated changes for the parameters are shown in the table below. Verify the data is stored correctly on the pacemaker device.

Table 2 Updated and New Programmable Parameters

Parameter	**Programmable Values**	**Increment**	**Nominal**
A or V Pulse Amplitude Regulated	Off, 0.1-5.0V	0.1V	5V
A or V Pulse Width	1-30 ms	1 ms	1 ms
A or V Sensitivity	0-5V	0.1V	N/A
Activity Threshold	V-Low, Low, Med-Low, Med, Med-High, High, V-High	N/A	Med
Reaction Time	10-50 sec	10 sec	30 sec

Response Factor	1-16	1	8
Recovery Time	2-16 min	1 min	5 min

Display Egram Data

When users choose to do so, the egram data for the ventricle and atrium should be displayed on the interface. This data should be received from the pacemaker via serial communication

DCM Design Decisions (Assignment 1)

While implementing the requirements outlined above for the DCM, the following design decisions were made.

Python & Imported Libraries/Packages

To create the DCM interface, the Python language was used to program it. This decision was made since Python has syntax that is easy to understand and work with. Additionally, it allowed for the utilization of various supported libraries and packages such as PySerial, Tkinter and matplotlib. These factors combined allowed for a generally smooth development of the DCM interface.

PySerial was used for determining if a device is connected to a computer port and what the device is.

Tkinter was used to create the graphical interface the user interacts with since it allowed for the implementation of all the requirements specified for the visual interface and the interactions. As well, it allowed for easy modification of the placements of buttons or text when prototyping.

Utilizing matplotlib made for a simple implementation for displaying the egram data when the user requests it.

Graphical Interface Implementation

The main goal for this DCM was to make it as user friendly and easy to use as possible whist not sacrificing the appearance and presentation of it.

When the DCM is run, the first window to pop up is the login/welcome window which has buttons for logging in and registering. Once successfully logged in, the window changes and now the user can select a pacing mode (VOO, AOO, VVI, AAI) or select to see the ventricle egram or atrium egram. When the user selects any of the modes, they will be able to now input the specified parameters for that mode and submit using the submit button

present. If the values are accepted, the user will see a message confirming the values are saved. If either of the egrams are selected on the home page, the animated egram plot will be shown that currently displays a plot of randomly generated data but will in the future display values from the pacemaker.

This interface utilizes some accessibility features to make the experience as easy as possible for people. On the login/welcome window, the “Login” and “Register” buttons are highlighted green for more visual contrast. Additionally, when the user is entering in values for the parameters for the different modes, if any values are invalid or aren’t accepted (not entering valid floats such as characters, or leaving some entries blank), an error message will pop up informing the user their inputs were not accepted. Once valid inputs are entered and the user clicks the submit button, a message appears in green text detailing the settings have been successfully saved. This is helps in preventing confusion related to if settings have been saved successfully and if the inputs were accepted.

As per one of the requirements, the contents of the windows adapt to the window size. They become more spaced out when the widow is enlarged, and they move closer together when the window is minimized.

Egram Data Implementation

To handle the egram data a class was created to manage and store the information (Class “EgramData”). It has attributes such as time stamps which store the values on the x-axis, voltages which store the voltage values on the y-axis, and a counter which increases by one each time a data point is added to the class. The counter is used to help animate the graph so when values are plotted the user can see the data currently being fed. To create and animate the plots the user will see when the ventricle or atrium egrams are selected on the interface, an additional class was created named “EgramPlotter”. This class uses randomly generated data and plots it, but in future assignments it will plot the voltages from the pacemaker.

Data Storage & Programmable Parameters

To store user’s login information, a CSV file is used, and the usernames and passwords are kept here. This makes it easy to validate the username and password, as well as ensure no more than 10 users can register and each is unique When a user is successfully logged in, they can select the desired mode and input values for the applicable parameters. If the inputs are accepted, i.e. if they are valid floats and no characters were inputted and a pacemaker board is connected, the values are then saved along with the mode to a CSV file in the format: Time, VOO, AOO, VVI, AAI, Values.

DCM Design Decisions (Assignment 2)

Through implementing the new requirements various new design decisions have been made. The additional code written for implementing serial communication is separated in a different document SerialCom.py, as well as the method to store the programmable parameters globalVars.py. This is to organize the code better and separate it into different sections. The contents of the files are accessed by importing them into the main file.

Graphical Interface Implementation

The revised interface still contains all of the original pages and general appearance though there have been improvements with the functionality. Previously, when navigating through this interface, as users traversed from the login screen to the homepage/select mode screen to the input settings page and back, each new environment would prompt a new window to popup which the user would have to exit by closing the window. This has been updated so that users can seamlessly navigate to the different pages and back by using back buttons implemented in the set parameters page and can return to the login screen from the homepage/select mode page via a logout button. The back buttons and logout button are coloured orange and red respectively. Another new update to the interface is another button added to the homepage which allows users to view the current pacing settings. On the homepage, the new pacing modes have also been added with their necessary pacing parameters.

Serial Communication

Serial Communication is implemented to transmit data from the DCM to the pacemaker using the PySerial library. This allows for an easy method to transmit the inputted settings to the pacemaker.

Programmable Parameters & Data Storage

User's login information is still being stored in a CSV file, and the valid inputted parameters are also still being saved to a CSV file but now instead of all the parameters being saved to the same CSV file, they are now saved to the user's personal CSV file that is created when registered.

All the possible programmable parameters are now stored using a dictionary. A function returning the dictionary (defaultParams()) assigns the names of they keys to be the parameters, and their values are initialized to default values. When a user submits valid inputs (i.e. inputs that fall within the ranges specified in Table 1 and 2), the dictionary is updated to reflect the desired settings, and these values are accessed when sending the parameters to the pacemaker device. The dictionary keeps these values until the user inputs new valid parameters and here the defaultParams() function is called to reset the

dictionary and it is then updated with the new settings. The defaultParams() function and the dictionary were implemented because it enabled an easy method for calling the pacing settings when sending data to the pacemaker and it ensured no invalid or potentially dangerous data would be sent to the device.

When settings are being sent to the pacemaker, they are first casted to be a specific type. The following table lists the parameter, the valid range it can be taken from the requirements, and what type it is casted to be.

Table 3 Programmable Parameters and their Data Type

Parameter	**Programmable Values**	**Data Type**
Lower Rate Limit	30-50 ppm 50 – 90 ppm 90-175 ppm	uint8 Range: 0-255
Upper Rate Limit	50-175 ppm	uint8 Range: 0-255
Maximum Sensor Rate	50-175 ppm	uint8 Range: 0-255
A or V Pulse Amplitude Regulated	Off, 0.5-5.0V	float
A or V Pulse Width	1-30 ms	uint8 Range: 0-255
A or V Sensitivity	0-5V	float
PVARP	150-500 ms	uint16 Range: 0-65535
Ventricular Refractory Period	150-500 ms	uint16 Range: 0-65535
Atrial Refractory Period	150-500 ms	uint16 Range: 0-65535
Activity Threshold	V-Low, Low, Med-Low, Med, Med-High, High, V-High	float
Reaction Time	10-50 sec	uint8 Range: 0-255
Response Factor	1-16	uint8 Range: 0-255
Recovery Time	2-16 min	uint8 Range: 0-255

Mode	A00, AAI, VOO, VVI, AOOR, AAIR, VOOR, VVIR	uint8 Range: 0-255

Egram Data Display

The data structure created to store and plot the egram data in part one is still being used, but now instead of plotting randomly generated values, the ventricle and atrium plots now show the actual voltages read from the pacemaker (pins A0 and A1).

Simulink Model (Assignment 1)

The full Simulink model is shown below. The model includes 3 main modules. The input parameters, the main logic that includes all the different pacemaker modes, and the output parameters/pins.

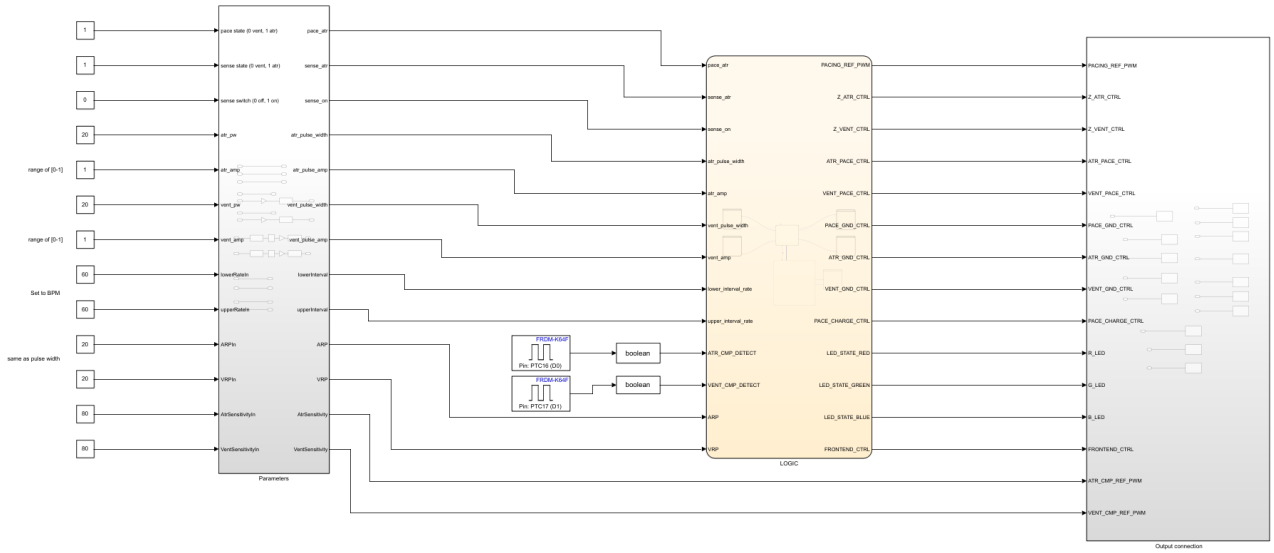


Figure 2: Simulink Model

The input parameters take all the necessary parameters as constants and either keeps them the same or goes through calculation blocks to convert them into a different variable. For example, the Lower Rate Limit constant, is converted into the lower rate interval by multiplying the reciprocal by 60,000. This converts the BPM into the amount of time between pulses.

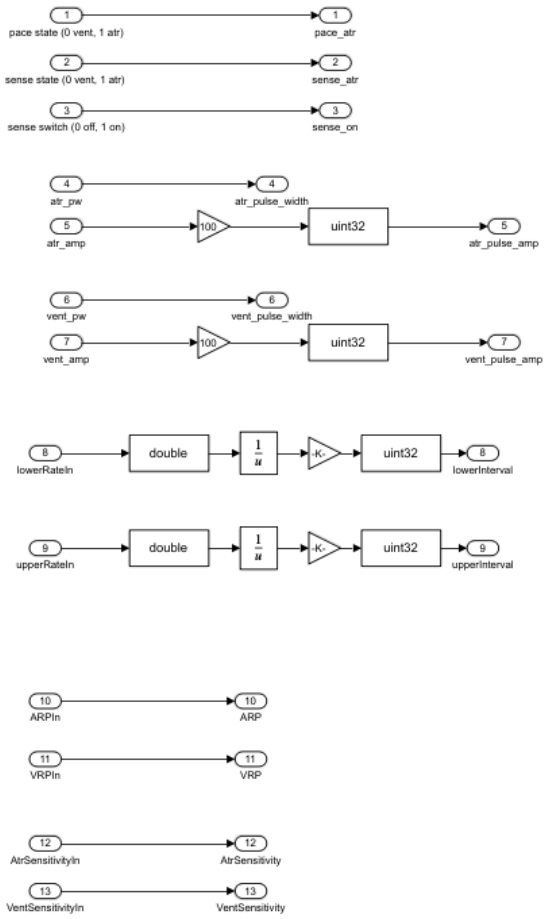


Figure 3: Input Parameters

The main logic has 5 charts inside. The initial state is there to provide the necessary grounding pins for safety reasons, and to set the initial amplitude for the selected mode. Each of the modes are accessed through conditions of specific variables. For example to get to VVI mode, the pace_atr variable has to be set to 0 (so that it is pacing ventricle), the sense_atr variable has to be set to 0 (so that it is sensing ventricle), and the sense_on variable has to be set to 1 so that it is able to inhibit the pulses.

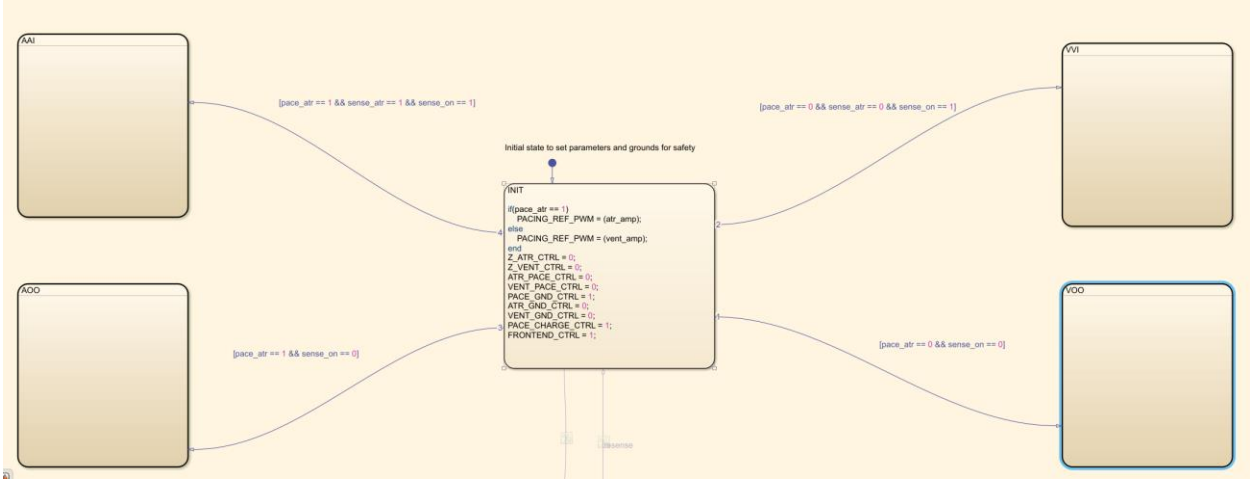


Figure 4: Simulink Main Logic

The AOO mode and VOO mode both use the same logic. They begin in the charging state with all of the correct pins set in the correct order. The only difference between the two is whether the ventricle or atrial pin is set to 1 for the corresponding mode. After the correct timing, the discharging block is activated, and the correct pins are set to discharge the capacitor. Then after the pulse width it begins to charge again.

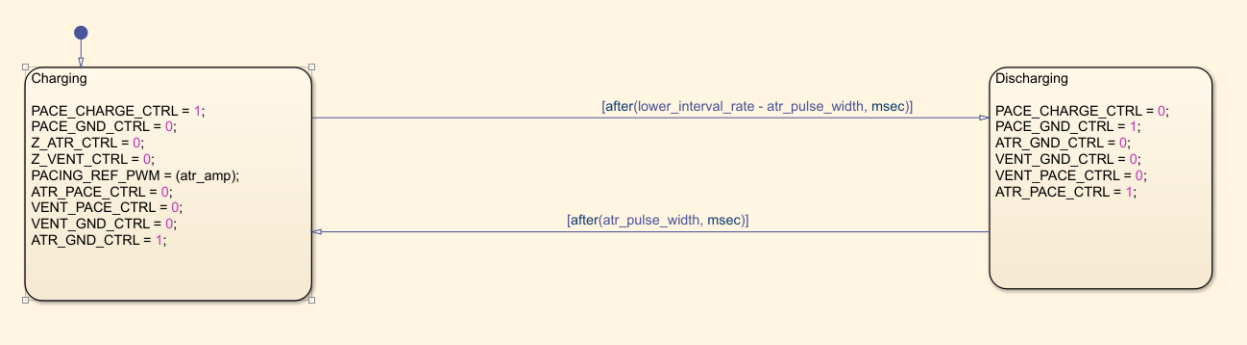


Figure 5: AOO Mode

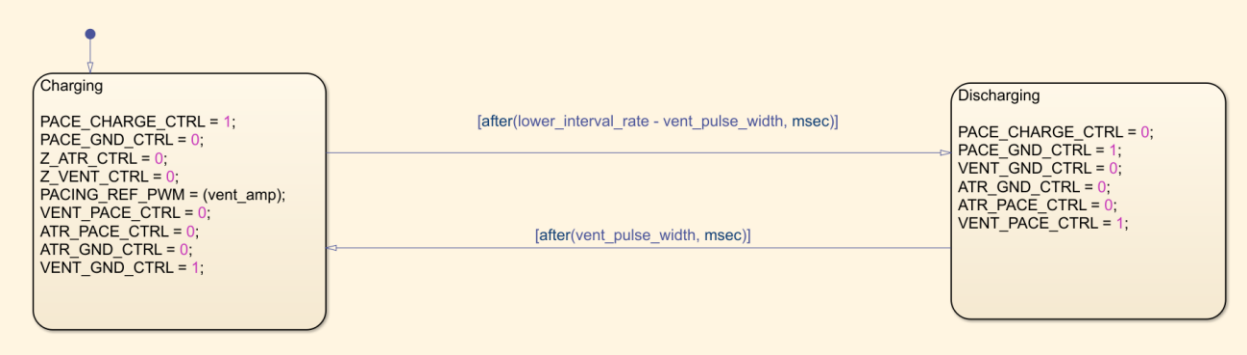


Figure 6: VOO Mode

The AAI and VVI modes work the same way but use their corresponding pins and variables. The main flow of the logic begins in the charging mode, then if a pulse is detected it returns to stay charging, however when a pulse isn't detected the pulse is discharged and returns back to charging afterwards.

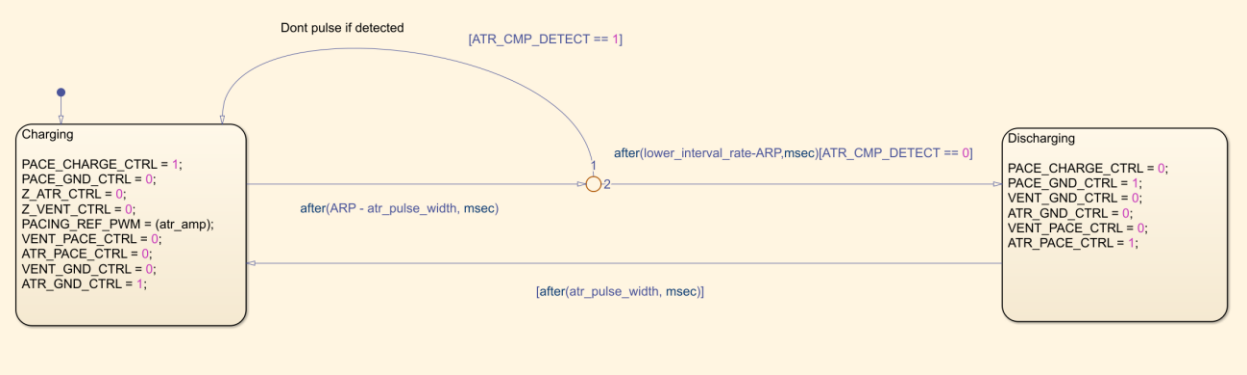


Figure 7: AAI Mode

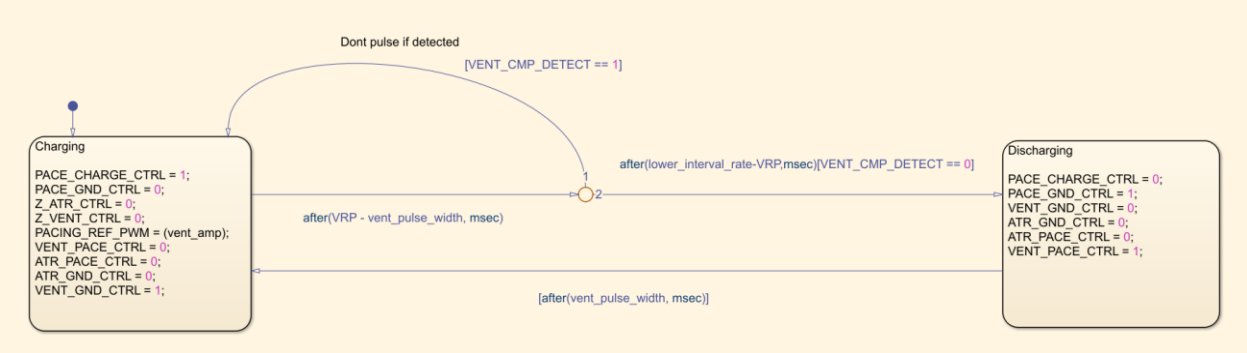


Figure 8: VVI Mode

The output pins take the correct pins and name them accordingly for easy access into the main logic.

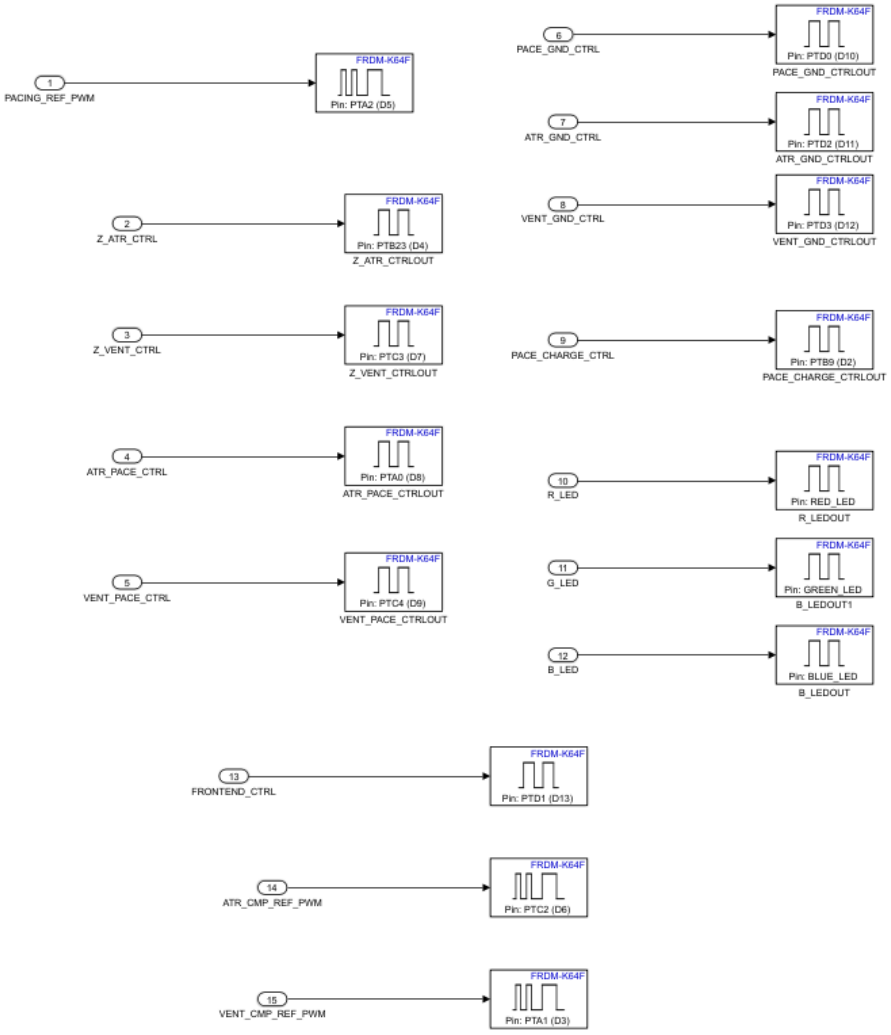


Figure 9: Output Pins

Simulink Model (Assignment 2)

We made several changes to the Simulink model for assignment 2. The main changes were implementing the serial communication, the Egram functionality, and the rate adaptive modes. The main logic also got updated for the bonus section by implementing double modes.

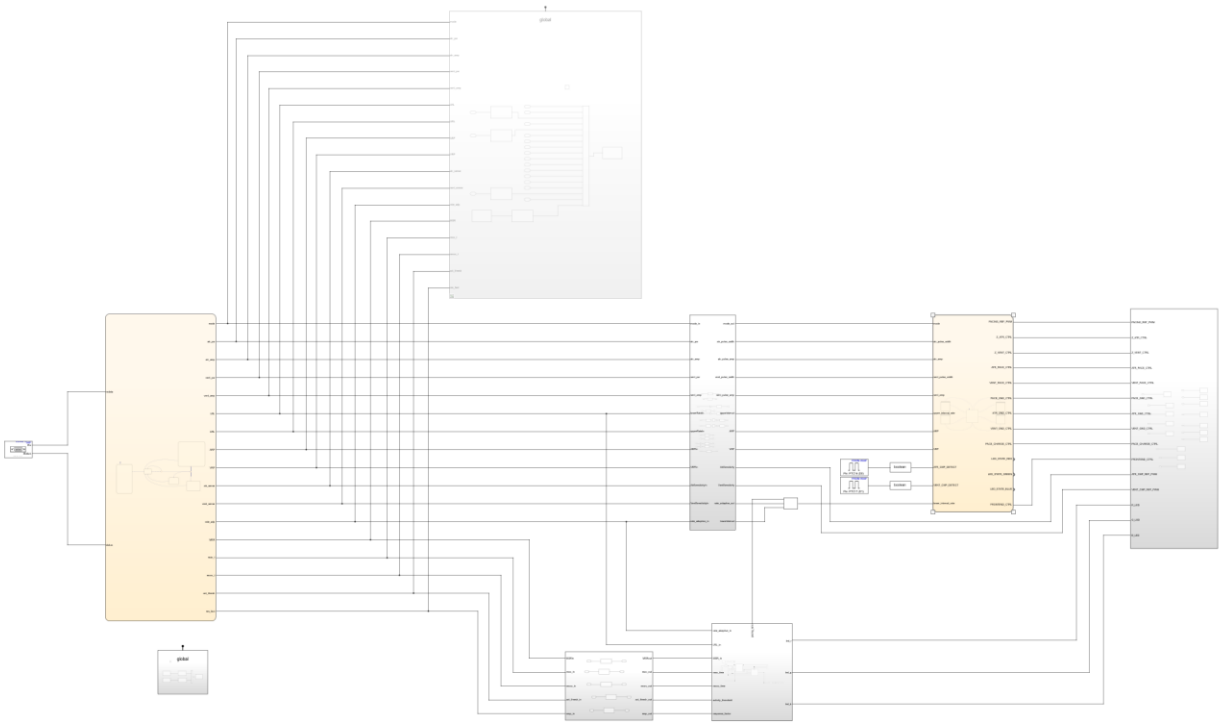


Figure 10: Updated Simulink Model

The updated model has the serial communication on the left, under it is the Egram function. The large subsystem above is connected to the each of the outputs from the serial chart, which is used to send back the data and was mainly used in testing. The rate adaptive logic is contained in the two subsystems in the bottom middle, the parameters are set in the left system, and the logic takes place in the right subsystem.

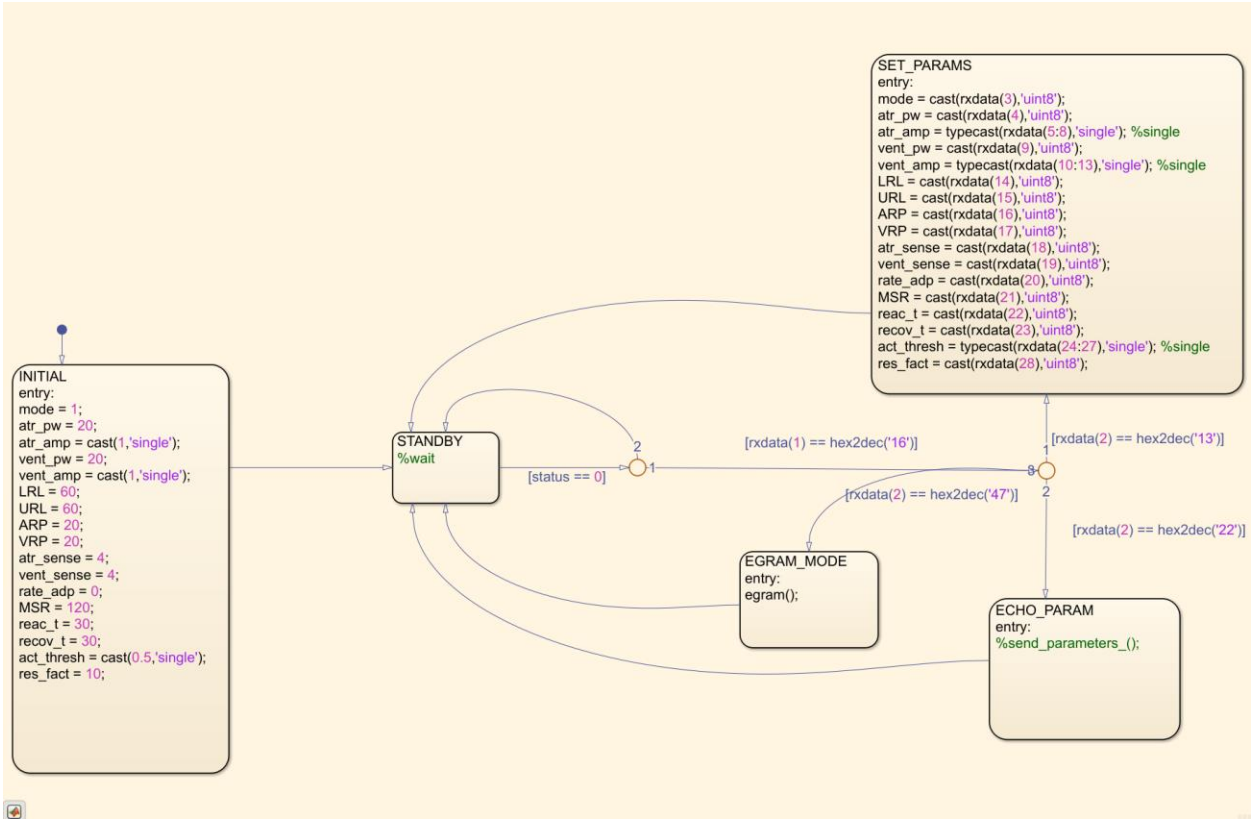


Figure 11: Serial Communication Logic

The serial communication logic begins by setting initial parameters to the default values, then it waits in standby for data to be sent. If data is sent to set parameters, then the variables will be updated corresponding to the specific order of bits. If data is sent to collect Egram data, then the Egram function will be called. The same is true for sending back parameters in the echo state.

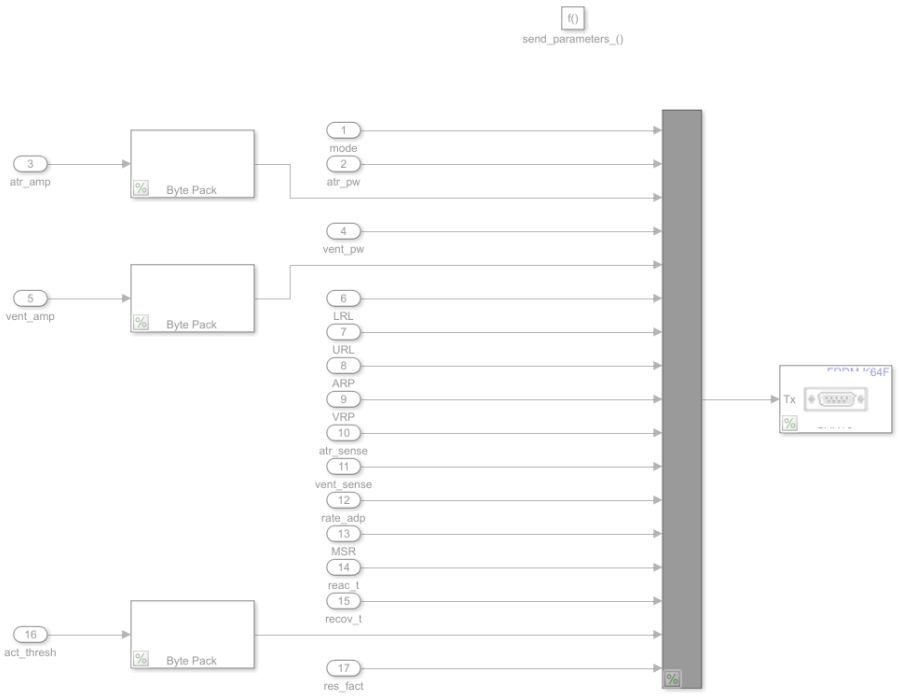


Figure 12: Send_Parameters Function

This function takes in the parameters and sends it back to the DCM through a serial transmit. Certain parameters had to be byte packed into a collection of uint8 data. This function was mainly used for testing.

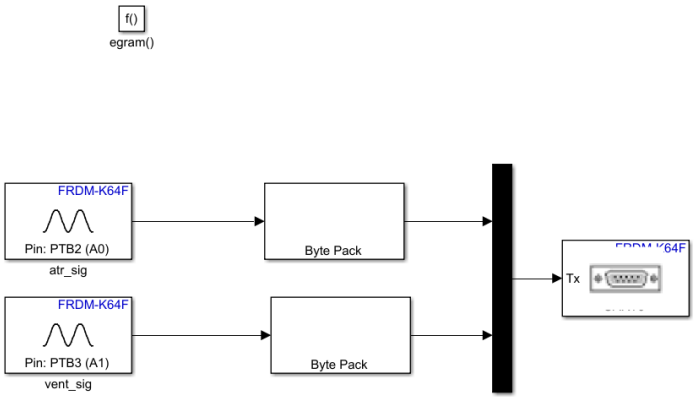


Figure 13: EGRAM Function

The function takes the data from the atrial and ventricle pins and byte packs them in order to send their data through a serial connection.

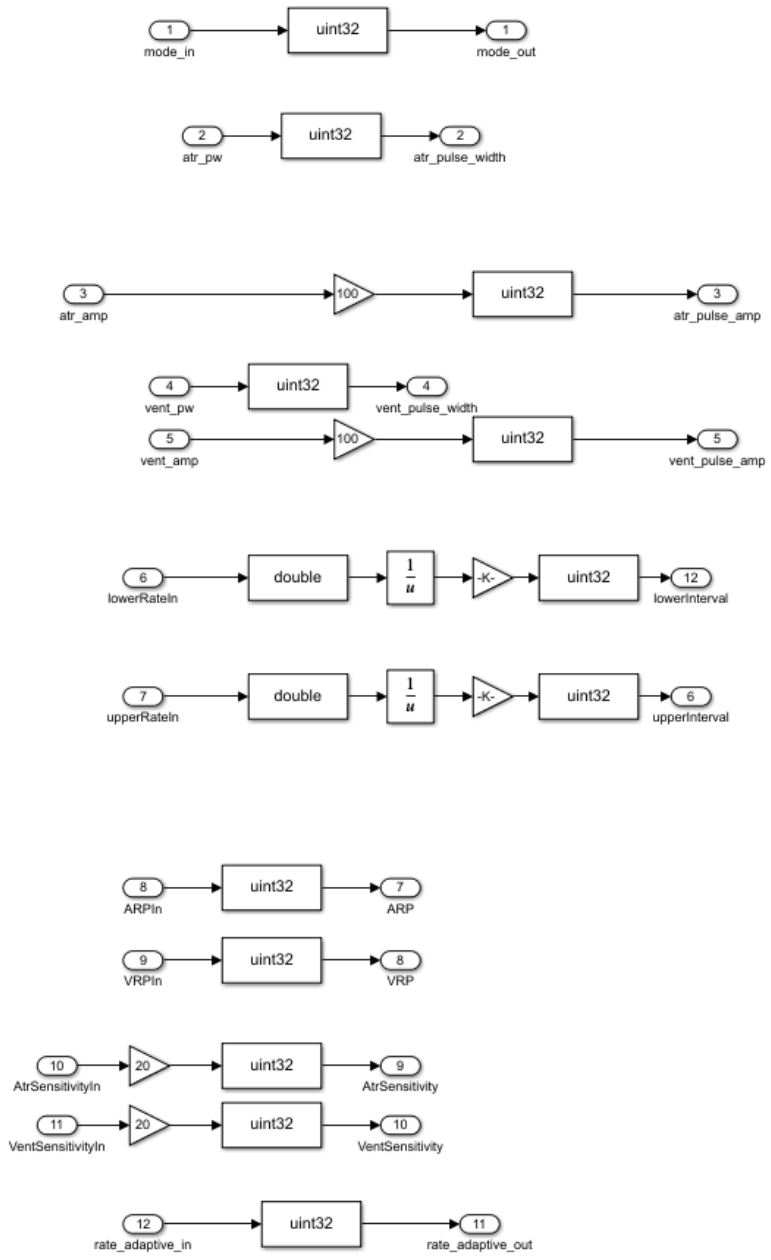


Figure 14: Updated Parameter Block

This function was updated to account for the new variables, and the data types were standardized into uint8 received, and uint32 being used in logic.

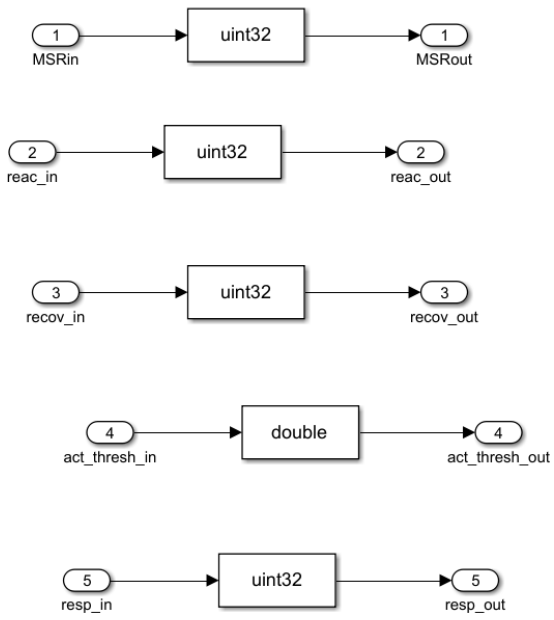


Figure 15: Rate Adaptive Parameters

The rate adaptive parameters are set to the correct data types here to be used in the rate adaptive logic.

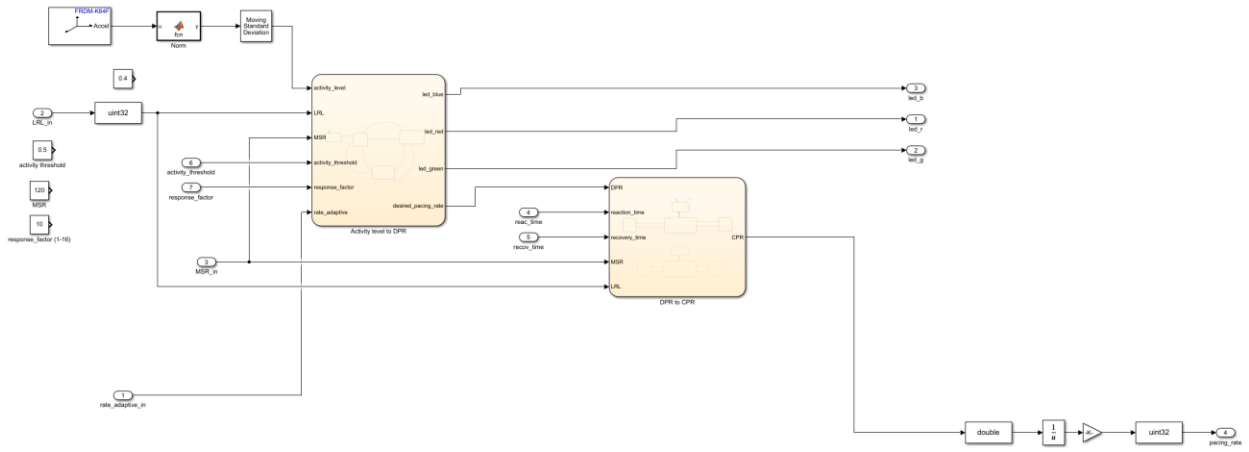


Figure 16: Rate Adaptive Logic

The rate adaptive logic takes in the accelerometer data and normalizes the x, y, and z components, then puts it through a moving standard deviation block to steady out the values. The accelerometer data is then put through the different mode logic which sets the desired pacing rate. Then the desired pacing rate goes through a system that updates the

current value at a specific rate per second to arrive at the desired pacing rate. The current rate is then sent to the main logic.

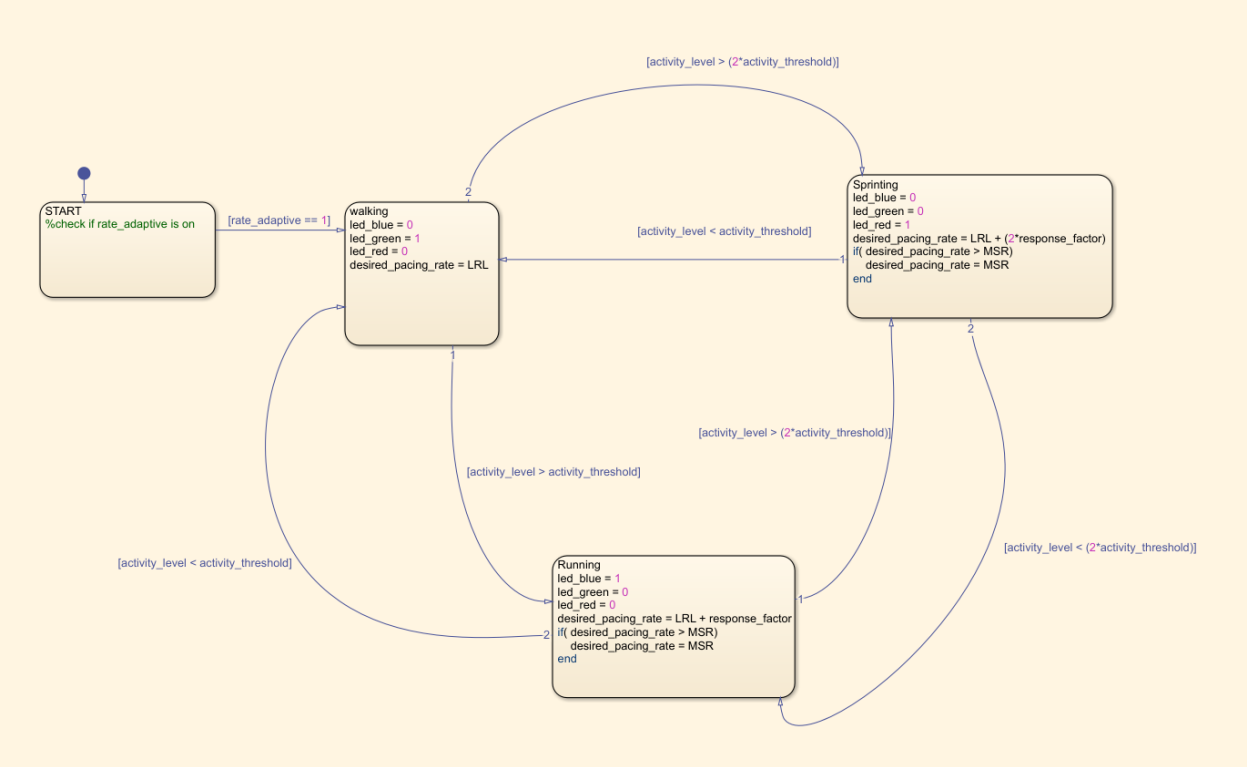


Figure 17: Accelerometer Data into Desired Pacing Rate

The logic to move through the states compares the activity level from the accelerometer to a specified activity threshold, and if the activity level reaches beyond it, it will move into a different state. To check this specific LEDs are turned on for each state.

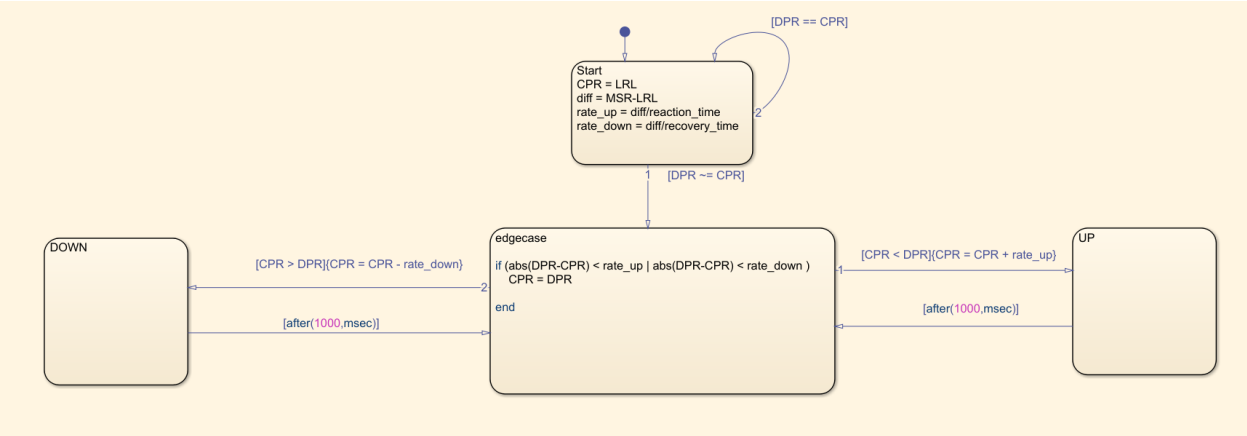


Figure 18: Desired Pacing Rate into Current Pacing Rate

The timing logic used the MSR and LRL parameters are used to set a specific rate that the desired pacing rate will increase/decrease by to get to the current pacing rate.

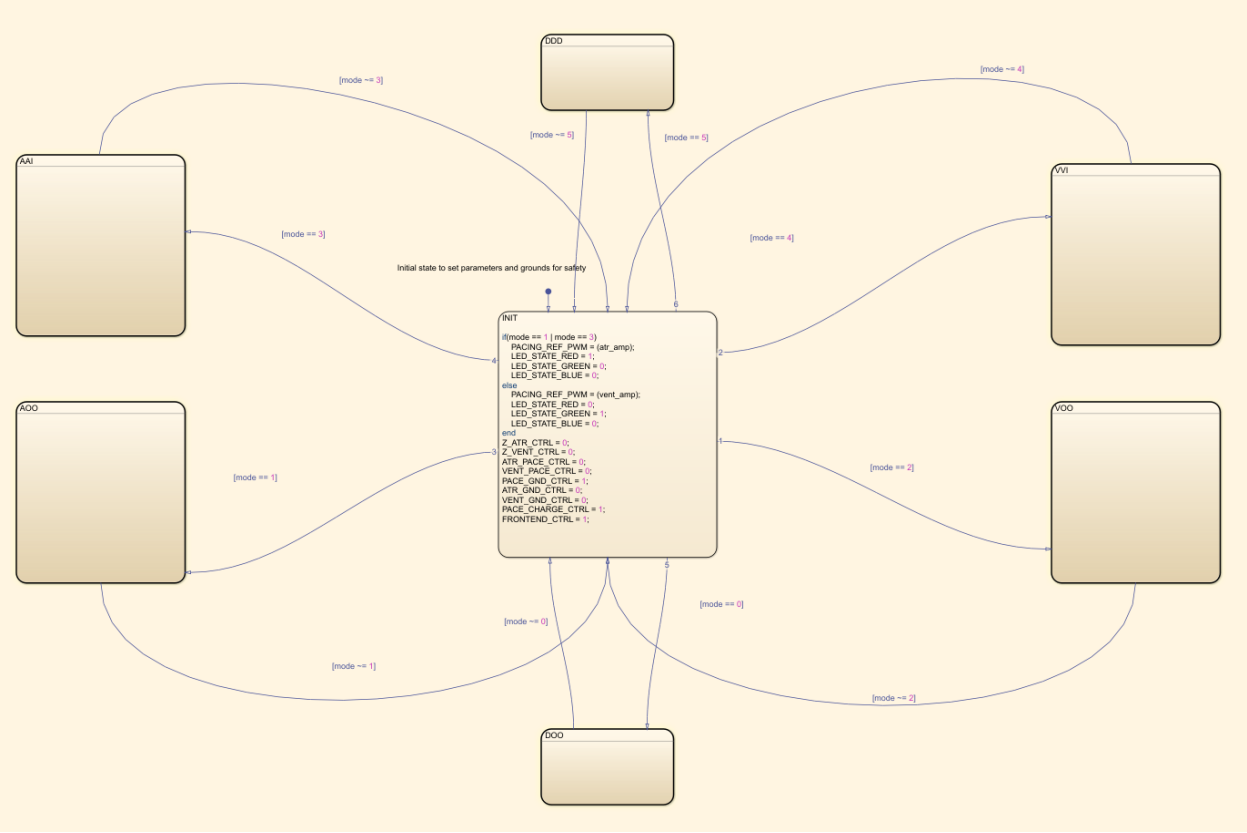


Figure 19: Updated Main Logic

The new logic contains modes for DOO and DDD

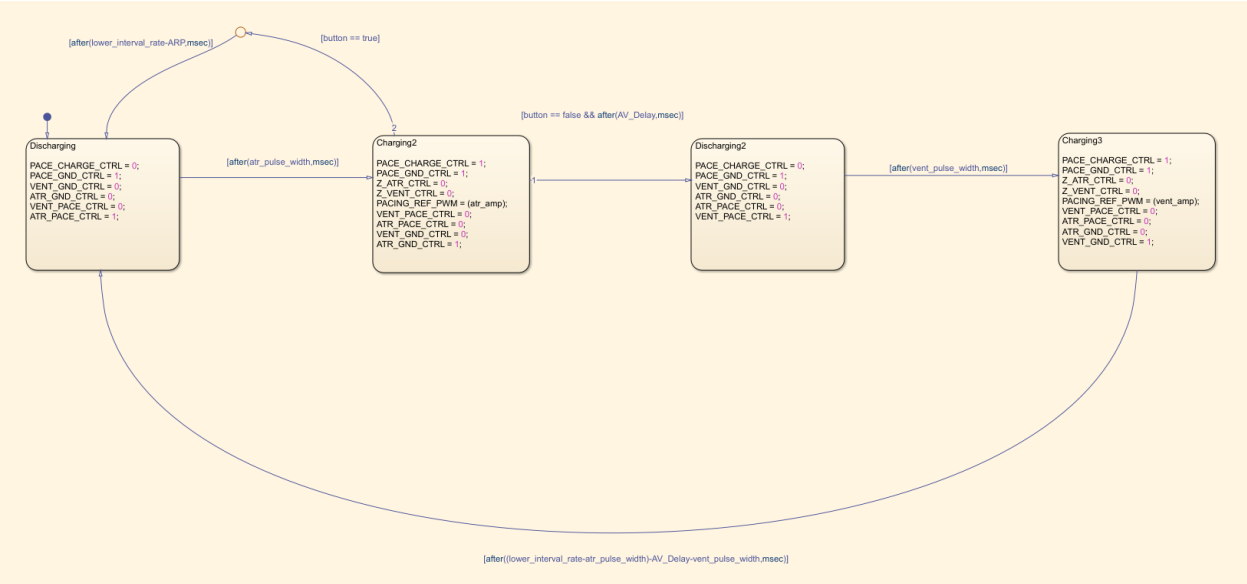


Figure 20: DOO Mode

The DOO mode uses and AV delay between atrial and ventricle pulses, and when the button is pressed the ventricle pulses will stop, but the atrial will continue.

DCM Code (Assignment 1)

The DCM code written in Python is as shown in the Figures below.

```python
import tkinter as tk
from tkinter import messagebox
import serial.tools.list_ports
import csv
import os
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random


'''
# This is where the data structures that describe the egram data and plots exists
# and sample functions to simulate graphs based on random data
'''

# Collections of egram data will be considered objects
class EgramData:

    # Construct object
    def __init__(self):
        # Manage labelling x axis
        self.counter = 0
        self.timestamps = []

        # Labelling y axis
        self.voltages = []

    # Setter. Take a datapoint and append to lists of past data
    def add_data(self, voltage):

        # Add timestamp and voltage to the lists
        self.timestamps.append(self.counter)
        self.voltages.append(voltage)
        self.counter+=1

        # Keep only the last 20 points
        if len(self.timestamps) > 20:
            self.timestamps = self.timestamps[-20:]
            self.voltages = self.voltages[-20:]

    # Getter for data
    def get_data(self):
        return self.timestamps, self.voltages
```

Figure 21 Python code featuring imported libraries and EgramData Class

```python
# Parent class of plots, each plot will have derived class in future assingments
class EgramPlotter:
    def __init__(self, title):
        # Initialize plot
        self.fig, self.ax = plt.subplots()
        self.title = title
        self.data = EgramData()

    # This generates each frame of animation
    def animate(self, i):
        # Generate random voltage data
        val = random.randrange(-10, 10, 1)
        voltage = round(val, 2)

        # Add that data
        self.data.add_data(voltage)

        # Collect data for plotting every frame
        timestamps, voltages = self.data.get_data()

        # Draw the plot every time
        self.ax.clear()
        self.ax.plot(timestamps, voltages)

        # Format plot
        self.ax.set_title(self.title)
        self.ax.set_ylabel('Voltage(V)')
        plt.xticks(rotation=45)


    def start_animation(self, interval):
        anim = animation.FuncAnimation(self.fig, self.animate, interval=interval)
        plt.show()
```

Figure 22 Python code featuring "EgramPlotter" class

```python
# Shell derived class for atrium plot, will be used in future assingments
class AtriumPlotter(EgramPlotter):
    def __init__(self):
        # Initialize from the parent class
        super().__init__("Atrium Electrogram")


# Shell derived class for ventricle plot, will be used in future assingments
class VentriclePlotter(EgramPlotter):
    def __init__(self):
        # Initialize from the parent class
        super().__init__("Ventricle Electrogram")


# Plot each graph
def plot_vent():
    ventricle_plotter = VentriclePlotter()
    ventricle_plotter.start_animation(100)


def plot_atrium():
    atrium_plotter = AtriumPlotter()
    atrium_plotter.start_animation(100)
```

Figure 23 Python code for Atrium and Ventricle plotter classes and functions to plot each graph

```
'''
# This section contains functions that handle the logic for initializing and storing users
'''

# Load in users to check logins
def load_users():

    # Find path to CSV containing login info
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    USER_FILE = os.path.join(FOLDER_PATH, 'users.csv')

    # Read each row of login info
    if os.path.exists(USER_FILE):
        with open(USER_FILE, mode='r', newline='') as file:
            reader = csv.reader(file)
            for row in reader:
                if len(row) == 2:

                    # Add entry to dictionary
                    name, password = row
                    users_db[name] = password


# Locally save valid login info
def save_user(name, password):

    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    USER_FILE = os.path.join(FOLDER_PATH, 'users.csv')

    # Write all current logins to CSV
    with open(USER_FILE, mode='w', newline='') as file:
        writer = csv.writer(file)
        for name, password in users_db.items():
            writer.writerow([name, password])
```

Figure 24 Python functions to check csv file to read and save login info

```
# Validation of user registration
def register_user():
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    USER_FILE = os.path.join(FOLDER_PATH, 'users.csv')

    # Count the locally stored users
    userNum = 0
    if os.path.exists(USER_FILE):

        for row in open(USER_FILE):
            userNum += 1

    # Pull user input
    name = entry_name.get()
    password = entry_password.get()

    # Check that both fields are filled
    if name and password:

        if userNum >= 10:
            messagebox.showerror("Error", "Too many users!")

        elif name in users_db:
            messagebox.showerror("Error", "User already exists!")

        else:
            users_db[name] = password
            save_user(name, password)
            messagebox.showinfo("Registration", "User registered successfully!")

    else:
        messagebox.showerror("Error", "Please fill out both fields.")
```

Figure 25 Python function to check if user can be registered

```python
# Check user login with saved users
def login_user():

    # Pull user input
    name = entry_name.get()
    password = entry_password.get()

    # Validate login info
    if name in users_db and users_db[name] == password:
        messagebox.showinfo("Login", f"Welcome, {name}!")
        mode_picker()

    else:
        messagebox.showerror("Error", "Incorrect username or password.")
```

Figure 26 Python function to login users

```python
'''
# This section contains the pacemaker discovery logic
'''

# Retrieve the unique device serial number
def get_serial(hwid):
    sections = hwid.split()
    for section in sections:
        if section.startswith("SER="):
            serial_number = section.split('=')[1]  # Get the value after "SER="
            return serial_number

# Check if there is a device connected and if so, return its info
def find_device():
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    USER_FILE = os.path.join(FOLDER_PATH, 'devices.csv')

    # Create tuple of ports and their connected info
    ports = list(serial.tools.list_ports.comports())

    for port in ports:
        # Check if pacemaker is connected
        if "JLink" in port.description:
            return get_serial(port.hwid)  # Use hwid for hardware ID containing the serial number
    return None
```

Figure 27 Python functions to get serial number from connected device and check if any device is connected

```python
# Determine whether the device is new
def save_device():
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    DEVICE_FILE = os.path.join(FOLDER_PATH, 'devices.csv')
    connected_device = find_device()

    if connected_device is None:
        print("No STM32 STLink device found.")
        return 0

    # Check if device is new
    if not os.path.exists(DEVICE_FILE):
        with open(DEVICE_FILE, mode='w', newline='') as file:  # Create the file if it doesn't exist
            pass  # Just create an empty file

    with open(DEVICE_FILE, mode='r', newline='') as file:
        reader = csv.reader(file)
        for row in reader:
            if connected_device in row:
                print("Device already saved.")
                return 1

    # If not, write the device info to the CSV
    with open(DEVICE_FILE, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([connected_device])
        print("Device saved.")
        messagebox.showinfo("Device", f"This device hasn't been connected yet!")
        return 2
```

Figure 28 Python function to save new device's info

```python
'''
# This section contains the mode picker screen, and connected device logic
'''

# Determine the first device connected, will be referenced to tell if different
# device is approached
def get_first_device():
    device = save_device()
    # Check if there is a device connected or not
    if device != 0:
        # Indicate the first device has been identified
        global first_device_flag
        first_device_flag = True
        id = find_device()
        print(id)
        return id


# Handle alert mechanism when a different device is approached
def alert_user():

    # Set true flag, indicate user has been alerted
    global device_compare_flag
    device_compare_flag = True
    messagebox.showinfo("Warning: A different pacemaker is approached than was previously interrogated")
```

Figure 29 Python functions to alert user if new device has been connected

```python
# Thread that updates connected device in background
def update_device_label():

    # Global var for first device, will be referenced for program runtime
    global first_device

    save_check = save_device()
    current_device = find_device()

    # Check if the device is not connected, new, or previously saved
    if save_check == 0:
        lbl_device.config(text="No device connected.")

    elif save_check == 1:
        lbl_device.config(text=f"Now communicating with device: \n{find_device()}")

        # if the first device hasn't been defined
        if first_device_flag == False:
            # Define first device
            first_device = get_first_device()

        # We only want the user to be alerted of different device once, so check
        # that there is a first device already, the current device is different,
        # and the user hasn't already been notified
        if (first_device_flag == True) and (current_device != first_device) and (device_compare_flag == False):
            alert_user()

        # When the first device is connected, get ready to flag different device
        if (first_device_flag == True) and (current_device == first_device):
            device_compare_flag = False

    else:
        lbl_device.config(text=f"Now communicating with device: \n{find_device()}")

        if first_device_flag == False:
            first_device = get_first_device()

        if (first_device_flag == True) and (current_device != first_device) and (device_compare_flag == False):
            alert_user()

        if (first_device_flag == True) and (current_device == first_device):
            device_compare_flag = False

    # Schedule the function to run recursively every 1000 ms
    root.after(1000, update_device_label)
```

Figure 30 Python device connection thread

```python
# Call this to create the mode selector window
def mode_picker():
    global lbl_device
    settings_window = tk.Toplevel(root)
    settings_window.title("Select mode")
    settings_window.geometry("450x350")  # Set appropriate window size

    # Adjust grid configuration to remove empty middle space
    settings_window.grid_columnconfigure(0, weight=1)  # Left margin
    settings_window.grid_columnconfigure(1, weight=1)  # VOO and VVI buttons
    settings_window.grid_columnconfigure(2, weight=1)  # AOO and AAI buttons
    settings_window.grid_columnconfigure(3, weight=1)  # Right margin

    settings_window.grid_rowconfigure(0, weight=1)  # Top margin row
    settings_window.grid_rowconfigure(2, weight=1)  # Spacer between buttons
    settings_window.grid_rowconfigure(4, weight=1)  # Spacer between pacing buttons and egrams
    settings_window.grid_rowconfigure(5, weight=1)  # Bottom margin row

    # Device label at the top
    lbl_device = tk.Label(settings_window, text="No device connected.", font=("Helvetica", 10, "bold"))
    lbl_device.grid(row=0, column=0, columnspan=4, pady=(5, 10))

    # Title label
    lbl_title = tk.Label(settings_window, text="Select Pacing Mode", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=1, column=0, columnspan=4, pady=(10, 20))  # Span all columns for centering

    # VOO button (left side)
    btn_voo = tk.Button(settings_window, text="VOO", command=open_voo_pacing_settings, width=12, height=2)
    btn_voo.grid(row=2, column=1, padx=(100 ,10), pady=10)

    # AOO button (right side)
    btn_aoo = tk.Button(settings_window, text="AOO", command=open_aoo_pacing_settings, width=12, height=2)
    btn_aoo.grid(row=2, column=2, padx=10, pady=10)

    # VVI button (left side)
    btn_vvi = tk.Button(settings_window, text="VVI", command=open_vvi_pacing_settings, width=12, height=2)
    btn_vvi.grid(row=3, column=1, padx=(100,10), pady=10)

    # AAI button (right side)
    btn_aai = tk.Button(settings_window, text="AAI", command=open_aai_pacing_settings, width=12, height=2)
    btn_aai.grid(row=3, column=2, padx=10, pady=10)
```

Figure 31 Python code for mode selection window

```python

    # Egram buttons placed in a single row at the bottom
    btn_vent_egram = tk.Button(settings_window, text="Ventricle Egram", command=plot_vent, width=12, height=2)
    btn_vent_egram.grid(row=5, column=1, padx=10, pady=10)

    btn_atrial_egram = tk.Button(settings_window, text="Atrium Egram", command=plot_atrium, width=12, height=2)
    btn_atrial_egram.grid(row=5, column=2, padx=10, pady=10)

    # Run this function recursively, update the connected device at interval
    update_device_label()
```

Figure 32 Python code for mode selection window cont.

```
'''
# This section handles the logic of storing the parameters to a CSV
'''

# Initialize CSV to store data
def initialize_csv_file():
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    PARAMETER_FILE = os.path.join(FOLDER_PATH, 'pacing_parameters.csv')
    with open(PARAMETER_FILE, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Time', 'VOO', 'AOO', 'VVI', 'AAI', 'Values'])
```

Figure 33 Python code to initialize csv file to store user inputs

```
# Call this to take all field inputs and write them to CSV
def save_settings(mode, lrl, url, amplitude, pulse_width, sensitivity=None, vrp=None, arp=None, pvarp=None, hysteresis=None, rate_smoothing=None):
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    PARAMETER_FILE = os.path.join(FOLDER_PATH, 'pacing_parameters.csv')

    # Open the CSV file in append mode
    with open(PARAMETER_FILE, mode='a', newline='') as file:
        writer = csv.writer(file)

        # Check for mode and write appropriate data to CSV
        if mode == 'VOO':
            writer.writerow([0, lrl, '', '', '','LRL'])
            writer.writerow([0, url, '', '', '','URL'])
            writer.writerow([0, amplitude, '', '', '','Amplitude'])
            writer.writerow([0, pulse_width, '', '', '','Pulse Width'])
        elif mode == 'AOO':
            writer.writerow([0, '', lrl, '', '','LRL'])
            writer.writerow([0, '', url, '', '','URL'])
            writer.writerow([0, '', amplitude, '', '','Amplitude'])
            writer.writerow([0, '', pulse_width, '', '','Pulse Width'])
        elif mode == 'VVI':
            writer.writerow([0, '', '', lrl, '','LRL'])
            writer.writerow([0, '', '', url, '','URL'])
            writer.writerow([0, '', '', amplitude, '','Amplitude'])
            writer.writerow([0, '', '', pulse_width, '','Pulse Width'])
            writer.writerow([0, '', '', sensitivity, '','Sensitivity'])
            writer.writerow([0, '', '', vrp, '','VRP'])
            writer.writerow([0, '', '', hysteresis, '','Hysteresis'])
            writer.writerow([0, '', '', rate_smoothing, '','Rate Smoothing'])
        elif mode == 'AAI':
            writer.writerow([0, '', '', '', lrl,'LRL'])
            writer.writerow([0, '', '', '', url,'URL'])
            writer.writerow([0, '', '', '', amplitude,'Amplitude'])
            writer.writerow([0, '', '', '', pulse_width,'Pulse Width'])
            writer.writerow([0, '', '', '', sensitivity,'Sensitivity'])
            writer.writerow([0, '', '', '', arp,'ARP'])
            writer.writerow([0, '', '', '', pvarp,'PVARP'])
            writer.writerow([0, '', '', '', hysteresis,'Hysteresis'])
            writer.writerow([0, '', '', '', rate_smoothing,'Rate Smoothing'])
```

Figure 34 Python function that writes inputted parameters to csv file

```
'''
# Create windows for user input for each setting
'''

# Call this to open the VOO settings to input
def open_voo_pacing_settings():
    settings_window = tk.Toplevel(root)
    settings_window.title("VOO Pacing Settings")
    settings_window.geometry("400x300")

    # Format the page grid
    settings_window.grid_columnconfigure(0, weight=1)  # Empty space on left
    settings_window.grid_columnconfigure(1, weight=0)  # Main elements
    settings_window.grid_columnconfigure(2, weight=0)
    settings_window.grid_columnconfigure(3, weight=1)  # Empty space on right
    settings_window.grid_rowconfigure(0, weight=1)  # Space above elements
    settings_window.grid_rowconfigure(7, weight=1)  # Space below elements

    # Put title
    lbl_title = tk.Label(settings_window, text="VOO Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(settings_window, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(settings_window)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(settings_window, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(settings_window)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Ventricle Amplitude
    lbl_va = tk.Label(settings_window, text="Ventricular Amplitude:")
    lbl_va.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_va = tk.Entry(settings_window)
    entry_va.grid(row=3, column=2, padx=10, pady=5)

    # Ventricle Pulse Width
    lbl_pw = tk.Label(settings_window, text="Ventricular Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(settings_window)
```

Figure 35 Python function to open VOO settings window

```
    entry_pw.grid(row=4, column=2, padx=10, pady=5)

    # Placeholder for submission status message
    lbl_status = tk.Label(settings_window, text="", fg="green")
    lbl_status.grid(row=6, column=1, columnspan=2, pady=5)

    # Input validation for submission of parameters
    def handle_submit():

        current_device = find_device()

        # Handle the cases that the entries aren't numbers and not all filled
        try:
            lrl = float(entry_lrl.get())
            url = float(entry_url.get())
            va = float(entry_va.get())
            pw = float(entry_pw.get())

        except ValueError:
            lbl_status.config(text="Please enter all fields as valid float numbers!", fg="red")
            return

        # Handle case when there's no board
        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="red")
            return

        # Save settings if all fields are numbers and a device is connected
        save_settings('VOO', lrl, url, va, pw)
        lbl_status.config(text="Settings have been saved!", fg="green")

        # After 3 seconds clear the message
        settings_window.after(3000, lambda: lbl_status.config(text=""))

    # Submit button
    btn_submit = tk.Button(settings_window, text="Submit", command=handle_submit)
    btn_submit.grid(row=5, column=1, columnspan=2, pady=10)
```

Figure 36 Python function to open VOO settings window cont.

```python
# Call this to open the AOO settings to input
def open_aoo_pacing_settings():
    settings_window = tk.Toplevel(root)
    settings_window.title("AOO Pacing Settings")
    settings_window.geometry("400x300")

    # Layout
    settings_window.grid_columnconfigure(0, weight=1)
    settings_window.grid_columnconfigure(1, weight=0)
    settings_window.grid_columnconfigure(2, weight=0)
    settings_window.grid_columnconfigure(3, weight=1)
    settings_window.grid_rowconfigure(0, weight=1)
    settings_window.grid_rowconfigure(6, weight=1)

    lbl_title = tk.Label(settings_window, text="AOO Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(settings_window, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(settings_window)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(settings_window, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(settings_window)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Atrial Amplitude
    lbl_aa = tk.Label(settings_window, text="Atrial Amplitude:")
    lbl_aa.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_aa = tk.Entry(settings_window)
    entry_aa.grid(row=3, column=2, padx=10, pady=5)

    # Atrial Pulse Width
    lbl_pw = tk.Label(settings_window, text="Atrial Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(settings_window)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)

    lbl_status = tk.Label(settings_window, text="", fg="green")
    lbl_status.grid(row=6, column=1, columnspan=2, pady=5)
```

Figure 37 Python function for AOO settings window

```python

    # Handle submission validation, refer to VOO function of same name
    def handle_submit():
        current_device = find_device()

        try:
            lrl = float(entry_lrl.get())
            url = float(entry_url.get())
            aa = float(entry_aa.get())
            pw = float(entry_pw.get())

        except ValueError:
            lbl_status.config(text="Please enter all fields as valid float numbers!", fg="red")
            return

        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="red")
            return

        save_settings('AOO', lrl, url, aa, pw)
        lbl_status.config(text="Settings have been saved!", fg="green")

        settings_window.after(3000, lambda: lbl_status.config(text=""))

    # Submit button
    btn_submit = tk.Button(settings_window, text="Submit", command=handle_submit)
    btn_submit.grid(row=5, column=1, columnspan=2, pady=10)
```

Figure 38 Python function for AOO settings window cont.

```python
def open_vvi_pacing_settings():
    settings_window = tk.Toplevel(root)
    settings_window.title("VVI Pacing Settings")
    settings_window.geometry("400x400")

    settings_window.grid_columnconfigure(0, weight=1)
    settings_window.grid_columnconfigure(1, weight=0)
    settings_window.grid_columnconfigure(2, weight=0)
    settings_window.grid_columnconfigure(3, weight=1)
    settings_window.grid_rowconfigure(0, weight=1)
    settings_window.grid_rowconfigure(9, weight=1)

    lbl_title = tk.Label(settings_window, text="VVI Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(settings_window, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(settings_window)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(settings_window, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(settings_window)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Ventricular Amplitude
    lbl_va = tk.Label(settings_window, text="Ventricular Amplitude:")
    lbl_va.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_va = tk.Entry(settings_window)
    entry_va.grid(row=3, column=2, padx=10, pady=5)

    # Ventricular Pulse Width
    lbl_pw = tk.Label(settings_window, text="Ventricular Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(settings_window)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)
```

Figure 39 Python function for VVI settings window

```python

    # Ventricular Sensitivity
    lbl_vs = tk.Label(settings_window, text="Ventricular Sensitivity:")
    lbl_vs.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_vs = tk.Entry(settings_window)
    entry_vs.grid(row=5, column=2, padx=10, pady=5)

    # VRP
    lbl_vrp = tk.Label(settings_window, text="VRP:")
    lbl_vrp.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_vrp = tk.Entry(settings_window)
    entry_vrp.grid(row=6, column=2, padx=10, pady=5)

    # Hysteresis
    lbl_hy = tk.Label(settings_window, text="Hysteresis:")
    lbl_hy.grid(row=7, column=1, sticky="e", padx=10, pady=5)
    entry_hy = tk.Entry(settings_window)
    entry_hy.grid(row=7, column=2, padx=10, pady=5)

    # Rate Smoothing
    lbl_rs = tk.Label(settings_window, text="Rate Smoothing:")
    lbl_rs.grid(row=8, column=1, sticky="e", padx=10, pady=5)
    entry_rs = tk.Entry(settings_window)
    entry_rs.grid(row=8, column=2, padx=10, pady=5)

    # Label to indicate settings have been saved (initially empty)
    lbl_status = tk.Label(settings_window, text="", fg="green")
    lbl_status.grid(row=10, column=1, columnspan=2, pady=5)

```

Figure 40 Python function for VVI settings window cont.

```
    # Handle submission validation, refer to VOO function of same name
    def handle_submit():

        current_device = find_device()

        try:
            lrl = float(entry_lrl.get())
            url = float(entry_url.get())
            va = float(entry_va.get())
            pw = float(entry_pw.get())
            vsens = float(entry_vs.get())
            vrp = float(entry_vrp.get())
            hy = float(entry_hy.get())
            rs = float(entry_rs.get())
        except ValueError:
            lbl_status.config(text="Please enter all fields as valid float numbers!", fg="red")
            return

        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="red")
            return

        save_settings('VVI', lrl, url, va, pw, vsens, vrp=vrp, hysteresis=hy, rate_smoothing=rs)
        lbl_status.config(text="Settings have been saved!", fg="green")

        settings_window.after(3000, lambda: lbl_status.config(text=""))

    # Submit button
    btn_submit = tk.Button(settings_window, text="Submit", command=handle_submit)
    btn_submit.grid(row=11, column=1, columnspan=2, pady=10)
```

Figure 41 Python function for VVI settings window cont.

```
def open_aai_pacing_settings():
    settings_window = tk.Toplevel(root)
    settings_window.title("AAI Pacing Settings")
    settings_window.geometry("500x500")

    settings_window.grid_columnconfigure(0, weight=1)
    settings_window.grid_columnconfigure(1, weight=0)
    settings_window.grid_columnconfigure(2, weight=0)
    settings_window.grid_columnconfigure(3, weight=1)
    settings_window.grid_rowconfigure(0, weight=1)
    settings_window.grid_rowconfigure(9, weight=1)

    lbl_title = tk.Label(settings_window, text="AAI Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(settings_window, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(settings_window)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(settings_window, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(settings_window)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Atrial Amplitude
    lbl_aa = tk.Label(settings_window, text="Atrial Amplitude:")
    lbl_aa.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_aa = tk.Entry(settings_window)
    entry_aa.grid(row=3, column=2, padx=10, pady=5)

    # Atrial Pulse Width
    lbl_pw = tk.Label(settings_window, text="Atrial Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(settings_window)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)
```

Figure 42 Python function for AAI settings window

```
736     # Atrial Sensitivity
737     lbl_as = tk.Label(settings_window, text="Atrial Sensitivity:")
738     lbl_as.grid(row=5, column=1, sticky="e", padx=10, pady=5)
739     entry_as = tk.Entry(settings_window)
740     entry_as.grid(row=5, column=2, padx=10, pady=5)
741
742     # ARP
743     lbl_arp = tk.Label(settings_window, text="ARP:")
744     lbl_arp.grid(row=6, column=1, sticky="e", padx=10, pady=5)
745     entry_arp = tk.Entry(settings_window)
746     entry_arp.grid(row=6, column=2, padx=10, pady=5)
747
748     # PVARP
749     lbl_pvarp = tk.Label(settings_window, text="PVARP:")
750     lbl_pvarp.grid(row=7, column=1, sticky="e", padx=10, pady=5)
751     entry_pvarp = tk.Entry(settings_window)
752     entry_pvarp.grid(row=7, column=2, padx=10, pady=5)
753
754     # Hysteresis
755     lbl_hy = tk.Label(settings_window, text="Hysteresis:")
756     lbl_hy.grid(row=8, column=1, sticky="e", padx=10, pady=5)
757     entry_hy = tk.Entry(settings_window)
758     entry_hy.grid(row=8, column=2, padx=10, pady=5)
759
760     # Rate Smoothing
761     lbl_rs = tk.Label(settings_window, text="Rate Smoothing:")
762     lbl_rs.grid(row=9, column=1, sticky="e", padx=10, pady=5)
763     entry_rs = tk.Entry(settings_window)
764     entry_rs.grid(row=9, column=2, padx=10, pady=5)
765
766     lbl_status = tk.Label(settings_window, text="", fg="green")
767     lbl_status.grid(row=11, column=1, columnspan=2, pady=5)
768
```

Figure 43 Python function for AAI settings window cont.

```
769     # Handle submission validation, refer to VOO function of same name
770     def handle_submit():
771         current_device = find_device()
772
773         try:
774             lrl = float(entry_lrl.get())
775             url = float(entry_url.get())
776             aa = float(entry_aa.get())
777             pw = float(entry_pw.get())
778             asens = float(entry_as.get())
779             arp = float(entry_arp.get())
780             pvarp = float(entry_pvarp.get())
781             hy = float(entry_hy.get())
782             rs = float(entry_rs.get())
783         except ValueError:
784             lbl_status.config(text="Please enter all fields as valid float numbers!", fg="red")
785             return
786
787         if current_device is None:
788             lbl_status.config(text="Please connect a board!", fg="red")
789             return
790
791
792         save_settings('AAI', lrl, url, aa, pw, asens, arp = arp, pvarp = pvarp, hysteresis = hy, rate_smoothing = rs)
793         lbl_status.config(text="Settings have been saved!", fg="green")
794
795         settings_window.after(3000, lambda: lbl_status.config(text=""))
796
797     # Submit button
798     btn_submit = tk.Button(settings_window, text="Submit", command=handle_submit)
799     btn_submit.grid(row=10, column=1, columnspan=2, pady=10)
800
```

Figure 44 Python function for AAI settings cont.

```
'''
# This section handles events that are bound to keys
'''

def on_login_enter_key(event):
    btn_login.invoke()  # Simulate a click on the login button
```

Figure 45 Python function to trigger login

```
'''
# Main application logic starts here
'''
# User database dictionary
users_db = {}

# Ready CSV to accept data
initialize_csv_file()

# Load users into dictionary
load_users()

# Initialize flags to handle device connecting
first_device_flag = False
device_compare_flag = False


# Main GUI window, initialize properties
root = tk.Tk()
root.title("Pacemaker DCM")
root.geometry("500x400")
root.configure(bg="#f0f0f0")

# Welcome message
welcome_message = tk.Label(root, text="\n\tWelcome\n", font=("Cambria", 24, "bold"), bg="#f0f0f0")
welcome_message.grid(row=0, column=0, columnspan=3, pady=(20, 0),)  # Centered by columnspan

welcome_message2 = tk.Label(root, text="\t Please Login or Register Below", font=("Cambria", 12), bg="#f0f0f0")
welcome_message2.grid(row=1, column=0, columnspan=3, pady=(0, 30))

# Name label and entry
lbl_name = tk.Label(root, text="Name:", bg="#f0f0f0")
lbl_name.grid(row=2, column=0, padx=(100, 0), pady=5)
entry_name = tk.Entry(root, width=30)
entry_name.grid(row=2, column=1, padx=(0, 0), pady=5)

```

Figure 46 Python code for main application logic

```
# Password label and entry
lbl_password = tk.Label(root, text="Password:", bg="#f0f0f0")
lbl_password.grid(row=3, column=0, padx=(100, 0), pady=5)
entry_password = tk.Entry(root, show="*", width=30)
entry_password.grid(row=3, column=1, padx=(0, 0), pady=5)

# Register button
btn_register = tk.Button(root, text="Register", command=register_user, bg="#4CAF50", fg="green", width=10, height = 1)
btn_register.grid(row=4, column=1, padx=(0, 0), pady=20, sticky="e")

# Login button with green outline
btn_login = tk.Button(root, text="Login", command=login_user, bg="white", fg="green", width=10, height = 1)
btn_login.grid(row=4, column=1, padx=(0, 0), pady=20, sticky="w")

# Bind the Enter key to trigger the login button
root.bind('<Return>', on_login_enter_key)

# Start the main event loop
root.mainloop()
```

Figure 47 Python code for main application logic cont.

DCM Code (Assignment 2)

The areas where Python code has been altered from the original code in Assignment 1 and the new code that has been added is shown in the figures below.

```python
import tkinter as tk
from tkinter import messagebox
import serial.tools.list_ports
import csv
import os
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

# Importing from separate files
import serialCom
from globalVars import defaultParams # default parameters function

# Setting default parameters for pacemaker
params = defaultParams()

# Global variables for entry fields and buttons
curr_user = None
entry_name = None
entry_password = None
btn_login = None
btn_register = None
lbl_device = None

# Finds current port
ports = list(serial.tools.list_ports.comports())
currPort = None
# To find port pacemaker is connected to:
for port in ports:
    print(port.description)

    if "JLink" in port.description:
        currPort = port.device
        break

if currPort:
    print(f"Device is connected to: {currPort}")
else:
    print("Device not found.")
```

Figure 48 Updated DCM code from Figure 21

```python
# Parent class of plots, each plot will have derived class in future assingments
class EgramPlotter:
    def __init__(self, title):
        # Initialize plot
        self.fig, self.ax = plt.subplots()
        self.title = title
        self.data = EgramData()

        # new code:
        self.ser = None
        self.window = tk.Tk()
        self.window.title(self.title)

        # Embed the Matplotlib figure in the Tkinter window
        self.canvas = FigureCanvasTkAgg(self.fig, master=self.window)
        self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=1)

        # Add the exit button
        self.exit_button = tk.Button(self.window, text="Exit", command=self.close_window)
        self.exit_button.pack(side=tk.BOTTOM)

    # Closes egram plot window
    def close_window(self):
        """Close the graph window and properly close the serial connection."""
        if self.ser and self.ser.is_open:
            self.ser.close()
            print("Serial connection closed.")
        self.window.destroy()

    # This generates each frame of animation
    def animate(self, i):
        # Generate random voltage data
        # val = random.randrange(-10, 10, 1)
        # voltage = round(val, 2)
        with serial.Serial(currPort, baudrate=115200,timeout=1) as ser:
            voltage = serialCom.get_plotData(ser,params)

        # Add that data
        self.data.add_data(voltage)

        # Collect data for plotting every frame
        timestamps, voltages = self.data.get_data()

        # Draw the plot every time
        self.ax.clear()
        self.ax.plot(timestamps, voltages)

        # Format plot
        self.ax.set_title(self.title)
        self.ax.set_ylabel('Voltage(V)')
        plt.xticks(rotation=45)
```

Figure 49 Updated EgramPlotter class, original displayed in Figure 22

```python
    def start_animation(self, interval):
        anim = animation.FuncAnimation(self.fig, self.animate, interval=interval)
        self.window.mainloop()
```

Figure 50 Updated function in EgramPlotter class, original shown in Figure 22

```python
def normalize(data, min_val=None, max_val=None):
    min_val = min_val if min_val is not None else min(data)
    max_val = max_val if max_val is not None else max(data)
    return [(v - min_val) / (max_val - min_val) for v in data] if max_val != min_val else [0.5 for _ in data]
```

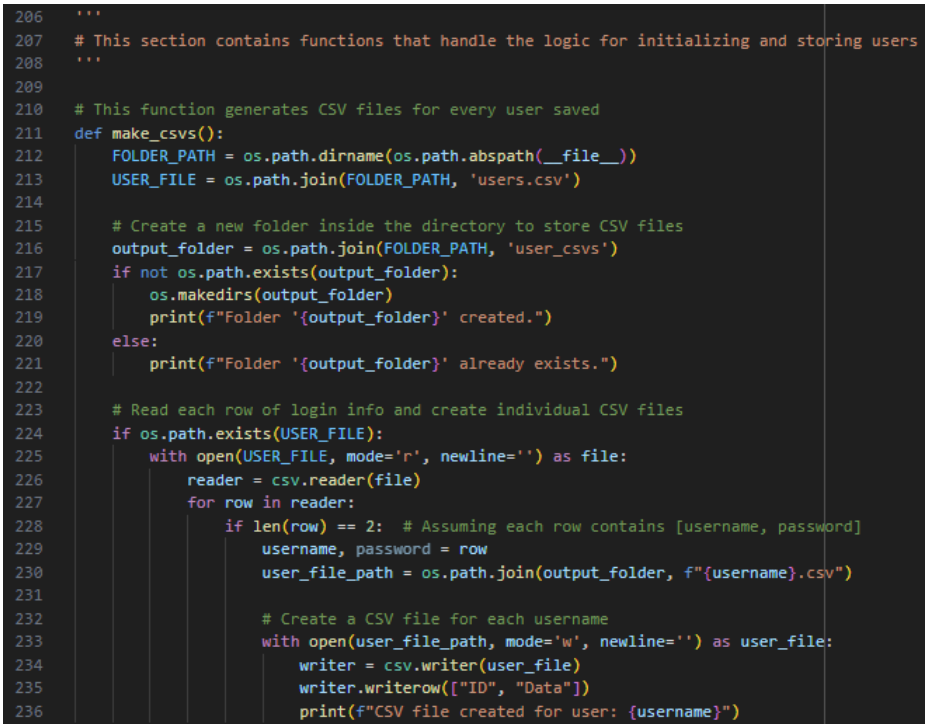
Figure 51 New normalize() function

```python
# Atrium instance of Egram
class AtriumPlotter(EgramPlotter):
    def __init__(self):
        # Initialize from the parent class
        super().__init__("Atrium Electrogram")

    def animate(self, i):
        with serial.Serial(currPort, baudrate=115200, timeout=1) as ser:
            voltages = serialCom.get_plotData(ser, params)

        if voltages is not None:
            voltage = voltages[0]  # Use unpacked[0] for atrium
            self.data.add_data(voltage)


        # Collect data for plotting every frame
        timestamps, voltages = self.data.get_data()

        # Draw the plot
        self.ax.clear()
        self.ax.plot(timestamps, voltages)

        # Format plot
        self.ax.set_title(self.title)
        self.ax.set_ylabel('Voltage(V)')
        self.ax.set_ylim(0, 1)  # Set static y-axis range
        plt.xticks(rotation=45)
```

Figure 52 Updated AtriumPlotter class, original shown in Figure 23

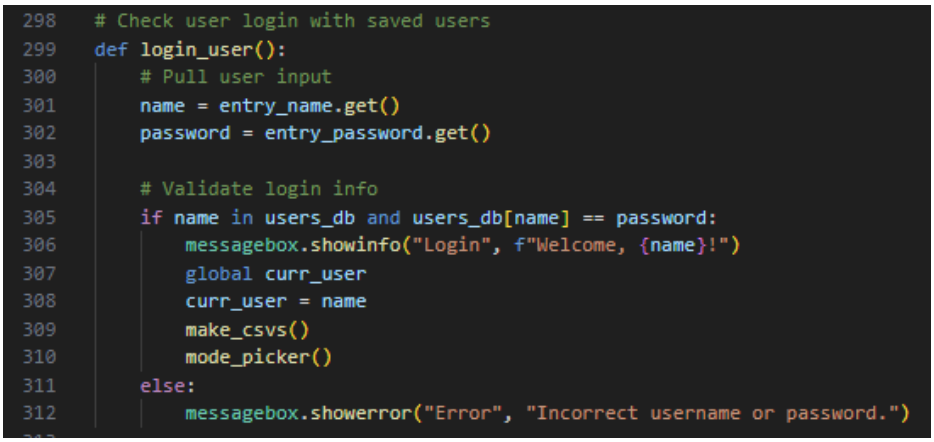
```python
# Ventricle instance of Egram
class VentriclePlotter(EgramPlotter):
    def __init__(self):
        # Initialize from the parent class
        super().__init__("Ventricle Electrogram")

    def animate(self, i):
        with serial.Serial(currPort, baudrate=115200, timeout=1) as ser:
            voltages = serialCom.get_plotData(ser, params)

        if voltages is not None:
            voltage = voltages[1]  # Use unpacked[1] for ventricle
            self.data.add_data(voltage)

        # Collect data for plotting every frame
        timestamps, voltages = self.data.get_data()

        # Draw the plot
        self.ax.clear()
        self.ax.plot(timestamps, voltages)

        # Format plot
        self.ax.set_title(self.title)
        self.ax.set_ylabel('Voltage(V)')
        plt.xticks(rotation=45)

```

Figure 53 Updated VentriclePlotter class, original shown in Figure 23



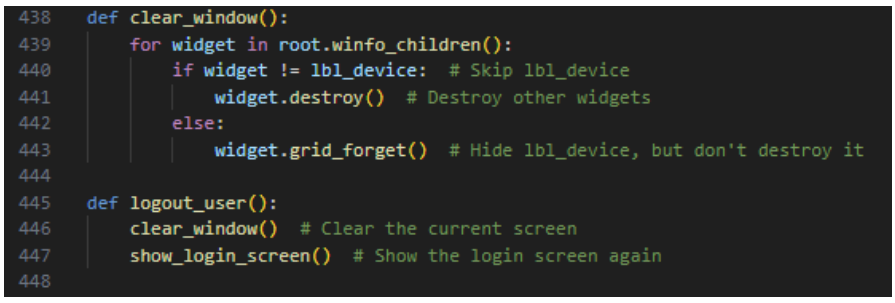
```
'''
# This section contains functions that handle the logic for initializing and storing users
'''

# This function generates CSV files for every user saved
def make_csvs():
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    USER_FILE = os.path.join(FOLDER_PATH, 'users.csv')

    # Create a new folder inside the directory to store CSV files
    output_folder = os.path.join(FOLDER_PATH, 'user_csvs')
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
        print(f"Folder '{output_folder}' created.")
    else:
        print(f"Folder '{output_folder}' already exists.")

    # Read each row of login info and create individual CSV files
    if os.path.exists(USER_FILE):
        with open(USER_FILE, mode='r', newline='') as file:
            reader = csv.reader(file)
            for row in reader:
                if len(row) == 2:  # Assuming each row contains [username, password]
                    username, password = row
                    user_file_path = os.path.join(output_folder, f"{username}.csv")

                    # Create a CSV file for each username
                    with open(user_file_path, mode='w', newline='') as user_file:
                        writer = csv.writer(user_file)
                        writer.writerow(["ID", "Data"])
                        print(f"CSV file created for user: {username}")
```

Figure 54 New function to make personal CSV files for users



```
# Check user login with saved users
def login_user():
    # Pull user input
    name = entry_name.get()
    password = entry_password.get()

    # Validate login info
    if name in users_db and users_db[name] == password:
        messagebox.showinfo("Login", f"Welcome, {name}!")
        global curr_user
        curr_user = name
        make_csvs()
        mode_picker()
    else:
        messagebox.showerror("Error", "Incorrect username or password.")
```

Figure 55 Updated login_user function, original shown in Figure 26



```
def clear_window():
    for widget in root.winfo_children():
        if widget != lbl_device:  # Skip lbl_device
            widget.destroy()  # Destroy other widgets
        else:
            widget.grid_forget()  # Hide lbl_device, but don't destroy it

def logout_user():
    clear_window()  # Clear the current screen
    show_login_screen()  # Show the login screen again

```

Figure 56 New functions to clear the window and logout user

```python
# Prompt that shows current settings
def display_current_settings():
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    SUBFOLDER = os.path.join(FOLDER_PATH, 'user_csvs')

    def get_user_csv_path(username):
        return os.path.join(SUBFOLDER, f"{username}.csv")

    # Get the path to the current user's CSV file
    PARAMETER_FILE = get_user_csv_path(curr_user)

    # Check if the CSV file exists
    if not os.path.exists(PARAMETER_FILE):
        messagebox.showerror("Error", "No saved settings found for the current user!")
        return

    try:
        # Read the last row from the CSV file
        with open(PARAMETER_FILE, mode='r') as file:
            reader = csv.reader(file)
            rows = list(reader)  # Read all rows into a list
            if not rows:  # Check if the file is empty
                messagebox.showerror("Error", "No settings found in the file!")
                return
            last_row = rows[-1]  # Get the most recent settings

        # Display the settings in a popup
        settings_display = "\n".join([f"{key}: {value}" for key, value in enumerate(last_row)])
        messagebox.showinfo("Current Settings", f"Most Recent Settings:\n\n{settings_display}")

    except Exception as e:
        messagebox.showerror("Error", f"Could not read settings: {str(e)}")
```

Figure 57 New function to display current pacing settings

```python
# Call this to create the mode selector window
def mode_picker():
    clear_window()
    global lbl_device
    root.title("Select mode")
    root.geometry("450x450")  # Set appropriate window size

    # Adjust grid configuration to remove empty middle space
    root.grid_columnconfigure(0, weight=1)  # Left margin
    root.grid_columnconfigure(1, weight=1)  # VOO and VVI buttons
    root.grid_columnconfigure(2, weight=1)  # AOO and AAI buttons
    root.grid_columnconfigure(3, weight=1)  # Right margin

    root.grid_rowconfigure(0, weight=1)  # Top margin row
    root.grid_rowconfigure(1, weight=1)  # Title row
    root.grid_rowconfigure(2, weight=1)  # Buttons row 1
    root.grid_rowconfigure(3, weight=1)  # Buttons row 2
    root.grid_rowconfigure(4, weight=1)  # Buttons row 3
    root.grid_rowconfigure(5, weight=1)  # Buttons row 4
    root.grid_rowconfigure(6, weight=1)  # Egram row
    root.grid_rowconfigure(7, weight=1)  # Bottom margin row

    # Device label at the top with darker r text
    lbl_device = tk.Label(root, text="No device connected.", font=("Helvetica", 10, "bold"), fg="r")
    lbl_device.grid(row=0, column=0, columnspan=4, pady=(8, 10))

    # Title label
    lbl_title = tk.Label(root, text="Select Pacing Mode", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=1, column=0, columnspan=4, pady=(10, 20))

    # VOO button (left side)
    btn_voo = tk.Button(root, text="VOO", command=open_voo_pacing_settings, width=12, height=2)
    btn_voo.grid(row=2, column=1, padx=10, pady=10)

    # AOO button (right side)
    btn_aoo = tk.Button(root, text="AOO", command=open_aoo_pacing_settings, width=12, height=2)
    btn_aoo.grid(row=2, column=2, padx=10, pady=10)

    # VVI button (left side)
    btn_vvi = tk.Button(root, text="VVI", command=open_vvi_pacing_settings, width=12, height=2)
    btn_vvi.grid(row=3, column=1, padx=10, pady=10)

    # AAI button (right side)
    btn_aai = tk.Button(root, text="AAI", command=open_aai_pacing_settings, width=12, height=2)
    btn_aai.grid(row=3, column=2, padx=10, pady=10)

    # VOOR button (left side)
    btn_voor = tk.Button(root, text="VOOR", command=open_voor_pacing_settings, width=12, height=2)
    btn_voor.grid(row=4, column=1, padx=10, pady=10)
```

Figure 58 Updated mode_picker() function, original shown in Figure 31

```python

    # AOOR button (right side)
    btn_aoor = tk.Button(root, text="AOOR", command=open_aoor_pacing_settings, width=12, height=2)
    btn_aoor.grid(row=4, column=2, padx=10, pady=10)

    # VVIR button (left side)
    btn_vvir = tk.Button(root, text="VVIR", command=open_vvir_pacing_settings, width=12, height=2)
    btn_vvir.grid(row=5, column=1, padx=10, pady=10)

    # AAIR button (right side)
    btn_aair = tk.Button(root, text="AAIR", command=open_aair_pacing_settings, width=12, height=2)
    btn_aair.grid(row=5, column=2, padx=10, pady=10)

    # Egram buttons placed in a single row at the bottom
    btn_vent_egram = tk.Button(root, text="Ventricle Egram", command=plot_vent, width=12, height=2)
    btn_vent_egram.grid(row=6, column=1, padx=10, pady=10)

    btn_atrial_egram = tk.Button(root, text="Atrium Egram", command=plot_atrium, width=12, height=2)
    btn_atrial_egram.grid(row=6, column=2, padx=10, pady=10)

    # Logout button
    btn_logout = tk.Button(root, text="Logout", command=logout_user, bg="r", fg="white", width=10, height=2)
    btn_logout.grid(row=7, column=0, columnspan=4, pady=(20, 20))  # Center below the mode selection buttons

    # User indication
    lbl_username = tk.Label(root, text=f"----  {curr_user} is logged in  ----", font=("Helvetica", 10, "bold"), fg="green")
    lbl_username.grid(row=8, column=0, columnspan=4, pady=(0, 20))

    # Current settings button
    btn_settings = tk.Button(root, text="Current Settings", command=display_current_settings, bg="#E49B0F", fg="white", width=15, height=2)
    btn_settings.grid(row=7, column=1, columnspan=4, padx=(0,230), pady=(20, 20))  # Center below the mode selection buttons

    # Run this function recursively, update the connected device at interval
    update_device_label()
```

Figure 59 Updated mode_picker() function cont., original shown in Figures 31 & 32

```
'''
# This section handles the logic of storing the parameters to a CSV
'''

# Writes the data to the appropriate CSV
def save_settings(mode, params):
    FOLDER_PATH = os.path.dirname(os.path.abspath(__file__))
    SUBFOLDER = os.path.join(FOLDER_PATH, 'user_csvs')

    def get_user_csv_path(username):
        return os.path.join(SUBFOLDER, f"{username}.csv")

    PARAMETER_FILE = get_user_csv_path(curr_user)

    # Open the CSV file in append mode
    with open(PARAMETER_FILE, mode='a', newline='') as file:
        writer = csv.writer(file)
        # Check for mode and write appropriate data to CSV
        if mode == 'VOO':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"],
                             params["url"], params["vent_amp"], params["vent_pw"]])
        elif mode == 'AOO':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"],
                             params["url"], params["atr_amp"], params["atr_pw"]])
        elif mode == 'VVI':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"],
                             params["url"], params["vent_amp"], params["vent_pw"],
                             params["vent_sens"], params["vrp"]])
        elif mode == 'AAI':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"],
                             params["url"], params["atr_amp"], params["atr_pw"],
                             params["atr_sens"], params["arp"], params["pvarp"]])
        elif mode == 'VOOR':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"],
                             params["url"], params["vent_amp"], params["vent_pw"], params["msr"],
                             params["act_thresh"], params["reaction_time"], params["response_fact"],
                             params["recovery_time"]])
        elif mode == 'AOOR':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"], params["url"],
                             params["atr_amp"], params["atr_pw"], params["msr"], params["act_thresh"],
                             params["reaction_time"], params["response_fact"], params["recovery_time"]])
        elif mode == 'VVIR':
            writer.writerow([0,params["mode"], params["rate_adapt"], params["lrl"], params["url"],
                             params["vent_amp"], params["vent_pw"], params["vent_sens"], params["vrp"],
                             params["msr"], params["act_thresh"], params["reaction_time"],
                             params["response_fact"], params["recovery_time"]])
        elif mode == 'AAIR':
            writer.writerow([0, params["mode"], params["rate_adapt"], params["lrl"], params["url"],
                             params["atr_amp"], params["atr_pw"], params["atr_sens"], params["arp"],
                             params["pvarp"], params["msr"], params["act_thresh"], params["reaction_time"],
                             params["response_fact"], params["recovery_time"]])
```

Figure 60 Updated save_settings() function, original shown in Figure 34

```
'''
# Create windows for user input for each setting
'''

# Call this to open the VOO settings to input
def open_voo_pacing_settings():
    clear_window()
    root.title("VOO Pacing Settings")
    root.geometry("400x300")

    # Format the page grid
    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(7, weight=1)

    # Put title
    lbl_title = tk.Label(root, text="VOO Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Ventricle Amplitude
    lbl_va = tk.Label(root, text="Ventricular Amplitude:")
    lbl_va.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_va = tk.Entry(root)
    entry_va.grid(row=3, column=2, padx=10, pady=5)

    # Ventricle Pulse Width
    lbl_pw = tk.Label(root, text="Ventricular Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(root)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)

    # Placeholder for submission status message
    lbl_status = tk.Label(root, text="", fg="green")
    lbl_status.grid(row=5, column=1, columnspan=2, pady=5)
```

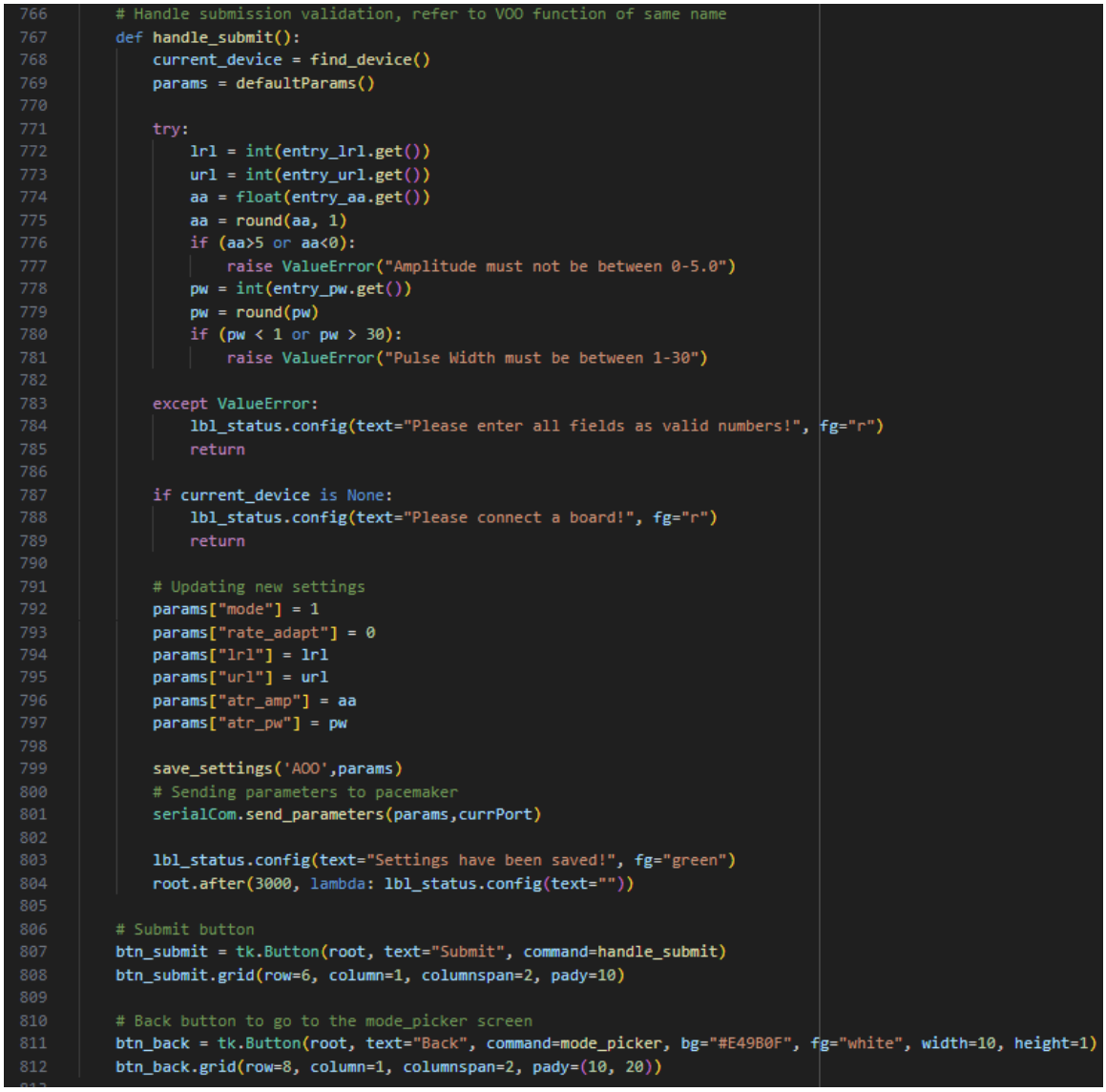
Figure 61 Updated VOO function, original shown in Figures 35 & 36

```
    # Input validation for submission of parameters
    def handle_submit():
        current_device = find_device()
        params = defaultParams()

        # Handle the cases that the entries aren't numbers and not all filled
        try:
            lrl = int(entry_lrl.get())
            url = int(entry_url.get())
            va = float(entry_va.get())
            va = round(va, 1)
            if (va>5 or va<0):
                raise ValueError("Amplitude must not be between 0-5.0")
            pw = int(entry_pw.get())
            pw = round(pw)
            if (pw < 1 or pw > 30):
                raise ValueError("Pulse Width must be between 1-30")

        except ValueError:
            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
            return

        # Handle case when there's no board
        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="r")
            return

        # updating new pacing settings
        params["mode"] = 2
        params["rate_adapt"] = 0
        params["lrl"] = lrl
        params["url"] = url
        params["vent_amp"] = va
        params["vent_pw"] = pw

        # Save settings if all fields are numbers and a device is connected
        save_settings('VOO', params)
        lbl_status.config(text="Settings have been saved!", fg="green")

        # sending parameters to pacemaker
        serialCom.send_parameters(params,currPort)

        # After 3 seconds clear the message
        root.after(3000, lambda: lbl_status.config(text=""))

    # Submit button
    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
    btn_submit.grid(row=6, column=1, columnspan=2, pady=10)

    # Back button to go to the mode_picker screen
    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
    btn_back.grid(row=8, column=1, columnspan=2, pady=(10, 20))
```

Figure 62 Updated open_voo_pacing_settings() function cont. original shown in Figures 35 & 36

```python
# Call this to open the AOO settings to input
def open_aoo_pacing_settings():
    clear_window()
    root.title("AOO Pacing Settings")
    root.geometry("400x300")

    # Layout
    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(6, weight=1)

    lbl_title = tk.Label(root, text="AOO Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Atrial Amplitude
    lbl_aa = tk.Label(root, text="Atrial Amplitude:")
    lbl_aa.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_aa = tk.Entry(root)
    entry_aa.grid(row=3, column=2, padx=10, pady=5)

    # Atrial Pulse Width
    lbl_pw = tk.Label(root, text="Atrial Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(root)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)

    lbl_status = tk.Label(root, text="", fg="green")
    lbl_status.grid(row=5, column=1, columnspan=2, pady=5)
```

Figure 63 Updated AOO pacing settings function, original shown in Figures 37 & 38



```
    # Handle submission validation, refer to VOO function of same name
    def handle_submit():
        current_device = find_device()
        params = defaultParams()

        try:
            lrl = int(entry_lrl.get())
            url = int(entry_url.get())
            aa = float(entry_aa.get())
            aa = round(aa, 1)
            if (aa>5 or aa<0):
                raise ValueError("Amplitude must not be between 0-5.0")
            pw = int(entry_pw.get())
            pw = round(pw)
            if (pw < 1 or pw > 30):
                raise ValueError("Pulse Width must be between 1-30")

        except ValueError:
            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
            return

        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="r")
            return

        # Updating new settings
        params["mode"] = 1
        params["rate_adapt"] = 0
        params["lrl"] = lrl
        params["url"] = url
        params["atr_amp"] = aa
        params["atr_pw"] = pw

        save_settings('AOO',params)
        # Sending parameters to pacemaker
        serialCom.send_parameters(params,currPort)

        lbl_status.config(text="Settings have been saved!", fg="green")
        root.after(3000, lambda: lbl_status.config(text=""))

    # Submit button
    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
    btn_submit.grid(row=6, column=1, columnspan=2, pady=10)

    # Back button to go to the mode_picker screen
    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
    btn_back.grid(row=8, column=1, columnspan=2, pady=(10, 20))
```

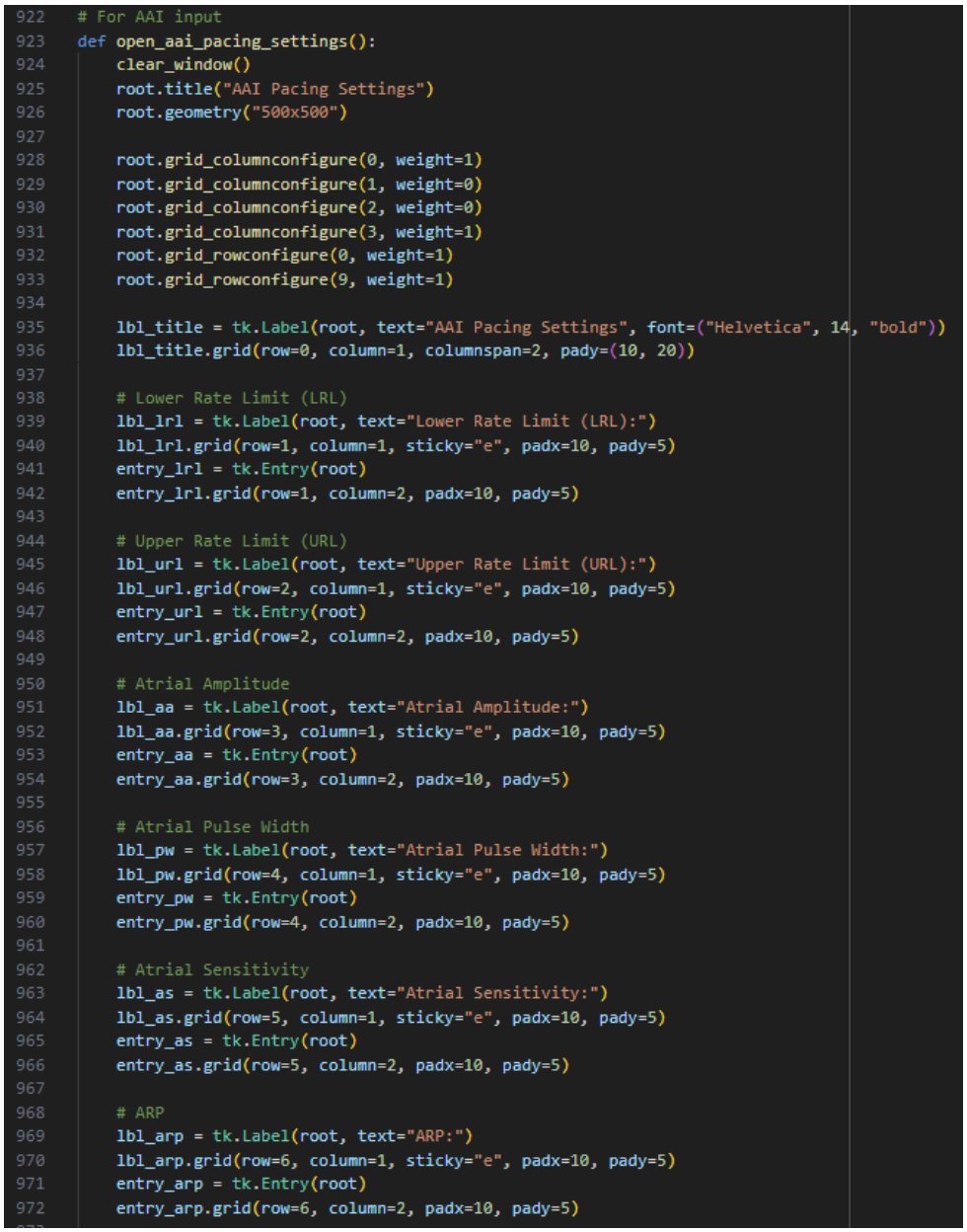
Figure 64 Updated AOO pacing settings fuction, original shown in Figures 37 & 38

```python
# For VVI input
def open_vvi_pacing_settings():
    clear_window()
    root.title("VVI Pacing Settings")
    root.geometry("400x400")

    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(9, weight=1)

    lbl_title = tk.Label(root, text="VVI Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Ventricular Amplitude
    lbl_va = tk.Label(root, text="Ventricular Amplitude:")
    lbl_va.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_va = tk.Entry(root)
    entry_va.grid(row=3, column=2, padx=10, pady=5)

    # Ventricular Pulse Width
    lbl_pw = tk.Label(root, text="Ventricular Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(root)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)

    # Ventricular Sensitivity
    lbl_vs = tk.Label(root, text="Ventricular Sensitivity:")
    lbl_vs.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_vs = tk.Entry(root)
    entry_vs.grid(row=5, column=2, padx=10, pady=5)

    # VRP
    lbl_vrp = tk.Label(root, text="VRP:")
    lbl_vrp.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_vrp = tk.Entry(root)
    entry_vrp.grid(row=6, column=2, padx=10, pady=5)

    # Label to indicate settings have been saved (initially empty)
    lbl_status = tk.Label(root, text="", fg="green")
    lbl_status.grid(row=7, column=1, columnspan=2, pady=5)
```

Figure 65 Updated VVI pacing settings function, original shown in Figures 39-41

```
    # Handle submission validation, refer to VOO function of same name
    def handle_submit():
        current_device = find_device()
        params = defaultParams()

        try:
            lrl = int(entry_lrl.get())
            url = int(entry_url.get())
            va = float(entry_va.get())
            va = round(va, 1)
            if (va>5 or va<0):
                raise ValueError("Amplitude must not be between 0-5.0")
            pw = int(entry_pw.get())
            pw = round(pw)
            if (pw < 1 or pw > 30):
                raise ValueError("Pulse Width must be between 1-30")
            vsens = int(entry_vs.get())
            if (vsens > 5 or vsens < 0):
                raise ValueError("Sensitivity must be between 0-5.0")
            vrp = int(entry_vrp.get())

        except ValueError:
            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
            return

        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="r")
            return

        params["mode"] = 4
        params["rate_adapt"] = 0
        params["lrl"] = lrl
        params["url"] = url
        params["vent_amp"] = va
        params["vent_pw"] = pw
        params["vent_sens"] = vsens
        params["vrp"] = vrp

        save_settings('VVI', params)
        serialCom.send_parameters(params,currPort)

        lbl_status.config(text="Settings have been saved!", fg="green")
        root.after(3000, lambda: lbl_status.config(text=""))

    # Submit button
    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
    btn_submit.grid(row=8, column=1, columnspan=2, pady=10)

    # Back button to go to the mode_picker screen
    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
    btn_back.grid(row=11, column=1, columnspan=2, pady=(10, 20))
```

Figure 66 Updated VVI pacing settings function cont., original shown in Figures 39-41



```
# For AAI input
def open_aai_pacing_settings():
    clear_window()
    root.title("AAI Pacing Settings")
    root.geometry("500x500")

    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(9, weight=1)

    lbl_title = tk.Label(root, text="AAI Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # Atrial Amplitude
    lbl_aa = tk.Label(root, text="Atrial Amplitude:")
    lbl_aa.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_aa = tk.Entry(root)
    entry_aa.grid(row=3, column=2, padx=10, pady=5)

    # Atrial Pulse Width
    lbl_pw = tk.Label(root, text="Atrial Pulse Width:")
    lbl_pw.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(root)
    entry_pw.grid(row=4, column=2, padx=10, pady=5)

    # Atrial Sensitivity
    lbl_as = tk.Label(root, text="Atrial Sensitivity:")
    lbl_as.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_as = tk.Entry(root)
    entry_as.grid(row=5, column=2, padx=10, pady=5)

    # ARP
    lbl_arp = tk.Label(root, text="ARP:")
    lbl_arp.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_arp = tk.Entry(root)
    entry_arp.grid(row=6, column=2, padx=10, pady=5)
```

Figure 67 Updated AAI pacing settings function, original shown in Figures 42-44

```
974         # PVARP
975         lbl_pvarp = tk.Label(root, text="PVARP:")
976         lbl_pvarp.grid(row=7, column=1, sticky="e", padx=10, pady=5)
977         entry_pvarp = tk.Entry(root)
978         entry_pvarp.grid(row=7, column=2, padx=10, pady=5)
979 
980         lbl_status = tk.Label(root, text="", fg="green")
981         lbl_status.grid(row=8, column=1, columnspan=2, pady=5)
982 
983         # Handle submission validation, refer to VOO function of same name
984         def handle_submit():
985             current_device = find_device()
986             params = defaultParams()
987 
988             try:
989                 lrl = int(entry_lrl.get())
990                 url = int(entry_url.get())
991                 aa = float(entry_aa.get())
992                 aa = round(aa, 1)
993                 if (aa>5 or aa<0):
994                     raise ValueError("Amplitude must not be between 0-5.0")
995                 pw = int(entry_pw.get())
996                 pw = round(pw)
997                 if (pw < 1 or pw > 30):
998                     raise ValueError("Pulse Width must be between 1-30")
999                 asens = int(entry_as.get())
1000                if (asens > 5 or asens < 0):
1001                    raise ValueError("Sensitivity must be between 0-5.0")
1002                arp = int(entry_arp.get())
1003                pvarp = int(entry_pvarp.get())
1004
1005            except ValueError:
1006                lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
1007                return
1008
1009            if current_device is None:
1010                lbl_status.config(text="Please connect a board!", fg="r")
1011                return
```

Figure 68 Updated AAI settings cont. original shown in Figures 42-44

```
1013            params["mode"] = 3
1014            params["rate_adapt"] = 0
1015            params["lrl"] = lrl
1016            params["url"] = url
1017            params["atr_amp"] = aa
1018            params["atr_pw"] = pw
1019            params["atr_sens"] = asens
1020            params["arp"] = arp
1021            params["pvarp"] = pvarp
1022
1023            save_settings('AAI',params)
1024            serialCom.send_parameters(params,currPort)
1025
1026            lbl_status.config(text="Settings have been saved!", fg="green")
1027            root.after(3000, lambda: lbl_status.config(text=""))
1028
1029        # Submit button
1030        btn_submit = tk.Button(root, text="Submit", command=handle_submit)
1031        btn_submit.grid(row=9, column=1, columnspan=2, pady=10)
1032
1033        # Back button to go to the mode_picker screen
1034        btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
1035        btn_back.grid(row=11, column=1, columnspan=2, pady=(10, 20))
```

Figure 69 Updates AAI pacing function cont., original shown in Figures 42-44

```python
# Call this to open the VOOR settings to input
def open_voor_pacing_settings():
    clear_window()
    root.title("VOOR Pacing Settings")
    root.geometry("400x600")

    # Format the page grid
    root.grid_columnconfigure(0, weight=1)  # Empty space on left
    root.grid_columnconfigure(1, weight=0)  # Main elements
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)  # Empty space on right
    root.grid_rowconfigure(0, weight=1)  # Space above elements
    root.grid_rowconfigure(7, weight=1)  # Space below elements

    # Put title
    lbl_title = tk.Label(root, text="VOOR Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # max sensor rate
    lbl_msr = tk.Label(root, text="Maximum Sensor Rate:")
    lbl_msr.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_msr = tk.Entry(root)
    entry_msr.grid(row=3, column=2, padx=10, pady=5)

    #activity threshold
    lbl_at = tk.Label(root, text="Activity Threshold:")
    lbl_at.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_at = tk.Entry(root)
    entry_at.grid(row=4, column=2, padx=10, pady=5)

    #reaction time
    lbl_rt = tk.Label(root, text="Reaction Time")
    lbl_rt.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_rt = tk.Entry(root)
    entry_rt.grid(row=5, column=2, padx=10, pady=5)
```

Figure 70 New function for VOOR pacing setting

```python
    #response factor
    lbl_rf = tk.Label(root, text="Response Factor")
    lbl_rf.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_rf = tk.Entry(root)
    entry_rf.grid(row=6, column=2, padx=10, pady=5)

    #recovery time
    lbl_rct = tk.Label(root, text="Recovery Time")
    lbl_rct.grid(row=7, column=1, sticky="e", padx=10, pady=5)
    entry_rct = tk.Entry(root)
    entry_rct.grid(row=7, column=2, padx=10, pady=5)

    # Ventricle Amplitude
    lbl_va = tk.Label(root, text="Ventricular Amplitude:")
    lbl_va.grid(row=8, column=1, sticky="e", padx=10, pady=5)
    entry_va = tk.Entry(root)
    entry_va.grid(row=8, column=2, padx=10, pady=5)

    # Ventricle Pulse Width
    lbl_pw = tk.Label(root, text="Ventricular Pulse Width:")
    lbl_pw.grid(row=9, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(root)
    entry_pw.grid(row=9, column=2, padx=10, pady=5)

    # Placeholder for submission status message
    lbl_status = tk.Label(root, text="", fg="green")
    lbl_status.grid(row=10, column=1, columnspan=2, pady=5)
```

Figure 71 New function for VOOR pacing setting cont.

```
    # Input validation for submission of parameters
    def handle_submit():
        current_device = find_device()
        params = defaultParams()

        # Handle the cases that the entries aren't numbers and not all filled
        try:
            lrl = int(entry_lrl.get())
            url = int(entry_url.get())
            msr = int(entry_msr.get())
            at = float(entry_at.get())
            rt = int(entry_rt.get())
            rf = int(entry_rf.get())
            rct = int(entry_rct.get())
            va = float(entry_va.get())
            va = round(va, 1)
            if (va>5 or va<0):
                raise ValueError("Amplitude must not be between 0-5.0")
            pw = int(entry_pw.get())
            pw = round(pw)
            if (pw < 1 or pw > 30):
                raise ValueError("Pulse Width must be between 1-30")

        except ValueError:
            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
            return

        # Handle case when there's no board
        if current_device is None:
            lbl_status.config(text="Please connect a board!", fg="r")
            return

        params["mode"] = 2
        params["rate_adapt"] = 1
        params["lrl"] = lrl
        params["url"] = url
        params["vent_amp"] = va
        params["vent_pw"] = pw
        params["msr"] = msr
        params["act_thresh"] = at
        params["reaction_time"] = rt
        params["response_fact"] = rf
        params["recovery_time"] = rct
        # Save settings if all fields are numbers and a device is connected
        save_settings('VOOR', params)
        serialCom.send_parameters(params,currPort)
        lbl_status.config(text="Settings have been saved!", fg="green")
        # After 3 seconds clear the message
        root.after(3000, lambda: lbl_status.config(text=""))
```

Figure 72 New function for VOOR cont.

```
    # Submit button
    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
    btn_submit.grid(row=11, column=1, columnspan=2, pady=10)

    # Back button to go to the mode_picker screen
    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
    btn_back.grid(row=13, column=1, columnspan=2, pady=(10, 20))
```

Figure 73 New function for VOOR cont.

```python
# Call this to open the AOOR settings to input
def open_aoor_pacing_settings():
    clear_window()
    root.title("AOOR Pacing Settings")
    root.geometry("400x500")

    # Layout
    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(6, weight=1)

    lbl_title = tk.Label(root, text="AOOR Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # max sensor rate
    lbl_msr = tk.Label(root, text="Maximum Sensor Rate:")
    lbl_msr.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_msr = tk.Entry(root)
    entry_msr.grid(row=3, column=2, padx=10, pady=5)

    #activity threshold
    lbl_at = tk.Label(root, text="Activity Threshold:")
    lbl_at.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_at = tk.Entry(root)
    entry_at.grid(row=4, column=2, padx=10, pady=5)

    #reaction time
    lbl_rt = tk.Label(root, text="Reaction Time")
    lbl_rt.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_rt = tk.Entry(root)
    entry_rt.grid(row=5, column=2, padx=10, pady=5)

    #response factor
    lbl_rf = tk.Label(root, text="Response Factor")
    lbl_rf.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_rf = tk.Entry(root)
    entry_rf.grid(row=6, column=2, padx=10, pady=5)
```

Figure 74 New function for AOOR

```python
    #recovery time
    lbl_rct = tk.Label(root, text="Recovery Time")
    lbl_rct.grid(row=7, column=1, sticky="e", padx=10, pady=5)
    entry_rct = tk.Entry(root)
    entry_rct.grid(row=7, column=2, padx=10, pady=5)

    # Atrial Amplitude
    lbl_aa = tk.Label(root, text="Atrial Amplitude:")
    lbl_aa.grid(row=8, column=1, sticky="e", padx=10, pady=5)
    entry_aa = tk.Entry(root)
    entry_aa.grid(row=8, column=2, padx=10, pady=5)

    # Atrial Pulse Width
    lbl_pw = tk.Label(root, text="Atrial Pulse Width:")
    lbl_pw.grid(row=9, column=1, sticky="e", padx=10, pady=5)
    entry_pw = tk.Entry(root)
    entry_pw.grid(row=9, column=2, padx=10, pady=5)

    lbl_status = tk.Label(root, text="", fg="green")
    lbl_status.grid(row=10, column=1, columnspan=2, pady=5)
```

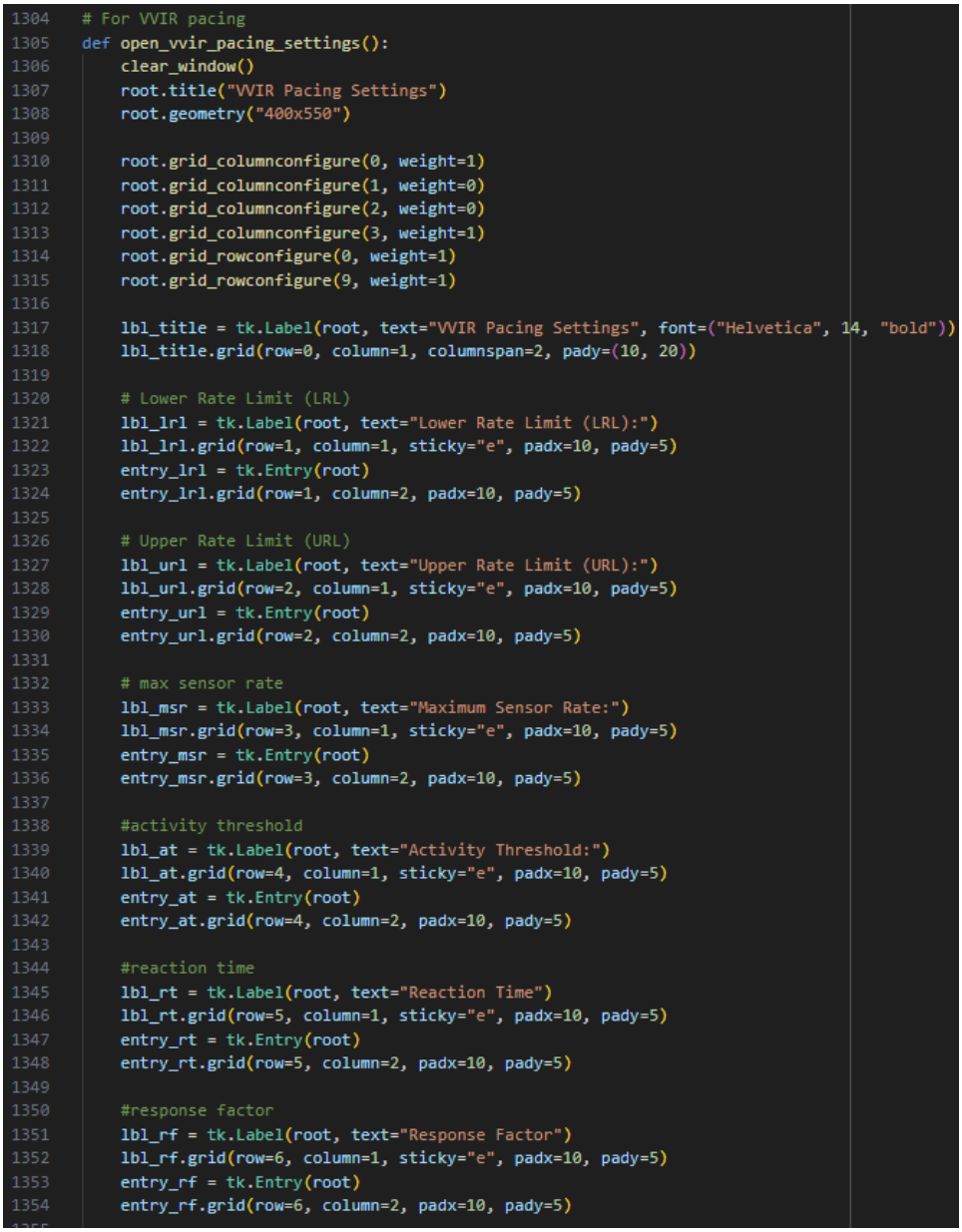
Figure 75 New function for AOOR cont.

```
1248    # Handle submission validation, refer to VOOR function of same name
1249    def handle_submit():
1250        current_device = find_device()
1251        params = defaultParams()
1252
1253        try:
1254            lrl = int(entry_lrl.get())
1255            url = int(entry_url.get())
1256            msr = int(entry_msr.get())
1257            at = float(entry_at.get())
1258            rt = int(entry_rt.get())
1259            rf = int(entry_rf.get())
1260            rct = int(entry_rct.get())
1261            aa = float(entry_aa.get())
1262            aa = round(aa, 1)
1263            if (aa>5 or aa<0):
1264                raise ValueError("Amplitude must not be between 0-5.0")
1265            pw = int(entry_pw.get())
1266            pw = round(pw)
1267            if (pw < 1 or pw > 30):
1268                raise ValueError("Pulse Width must be between 1-30")
1269
1270        except ValueError:
1271            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
1272            return
1273
1274        if current_device is None:
1275            lbl_status.config(text="Please connect a board!", fg="r")
1276            return
1277
1278        params["mode"] = 1
1279        params["rate_adapt"] = 1
1280        params["lrl"] = lrl
1281        params["url"] = url
1282        params["atr_amp"] = aa
1283        params["atr_pw"] = pw
1284        params["msr"] = msr
1285        params["act_thresh"] = at
1286        params["reaction_time"] = rt
1287        params["response_fact"] = rf
1288        params["recovery_time"] = rct
1289
1290        save_settings('AOOR', params)
1291        serialCom.send_parameters(params,currPort)
1292
1293        lbl_status.config(text="Settings have been saved!", fg="green")
1294        root.after(3000, lambda: lbl_status.config(text=""))
1295
1296    # Submit button
1297    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
1298    btn_submit.grid(row=11, column=1, columnspan=2, pady=10)
```

Figure 76 New function for AOOR cont.

```
1300    # Back button to go to the mode_picker screen
1301    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
1302    btn_back.grid(row=13, column=1, columnspan=2, pady=(10, 20))
```

Figure 77 New function for AOOR cont.



```
# For VVIR pacing
def open_vvir_pacing_settings():
    clear_window()
    root.title("VVIR Pacing Settings")
    root.geometry("400x550")

    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(9, weight=1)

    lbl_title = tk.Label(root, text="VVIR Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    # max sensor rate
    lbl_msr = tk.Label(root, text="Maximum Sensor Rate:")
    lbl_msr.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_msr = tk.Entry(root)
    entry_msr.grid(row=3, column=2, padx=10, pady=5)

    #activity threshold
    lbl_at = tk.Label(root, text="Activity Threshold:")
    lbl_at.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_at = tk.Entry(root)
    entry_at.grid(row=4, column=2, padx=10, pady=5)

    #reaction time
    lbl_rt = tk.Label(root, text="Reaction Time")
    lbl_rt.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_rt = tk.Entry(root)
    entry_rt.grid(row=5, column=2, padx=10, pady=5)

    #response factor
    lbl_rf = tk.Label(root, text="Response Factor")
    lbl_rf.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_rf = tk.Entry(root)
    entry_rf.grid(row=6, column=2, padx=10, pady=5)
```

Figure 78 New function for VVIR mode

```
1356        #recovery time
1357        lbl_rct = tk.Label(root, text="Recovery Time")
1358        lbl_rct.grid(row=7, column=1, sticky="e", padx=10, pady=5)
1359        entry_rct = tk.Entry(root)
1360        entry_rct.grid(row=7, column=2, padx=10, pady=5)
1361
1362        # Ventricular Amplitude
1363        lbl_va = tk.Label(root, text="Ventricular Amplitude:")
1364        lbl_va.grid(row=8, column=1, sticky="e", padx=10, pady=5)
1365        entry_va = tk.Entry(root)
1366        entry_va.grid(row=8, column=2, padx=10, pady=5)
1367
1368        # Ventricular Pulse Width
1369        lbl_pw = tk.Label(root, text="Ventricular Pulse Width:")
1370        lbl_pw.grid(row=9, column=1, sticky="e", padx=10, pady=5)
1371        entry_pw = tk.Entry(root)
1372        entry_pw.grid(row=9, column=2, padx=10, pady=5)
1373
1374        # Ventricular Sensitivity
1375        lbl_vs = tk.Label(root, text="Ventricular Sensitivity:")
1376        lbl_vs.grid(row=10, column=1, sticky="e", padx=10, pady=5)
1377        entry_vs = tk.Entry(root)
1378        entry_vs.grid(row=10, column=2, padx=10, pady=5)
1379
1380        # VRP
1381        lbl_vrp = tk.Label(root, text="VRP:")
1382        lbl_vrp.grid(row=11, column=1, sticky="e", padx=10, pady=5)
1383        entry_vrp = tk.Entry(root)
1384        entry_vrp.grid(row=11, column=2, padx=10, pady=5)
1385
1386        # Label to indicate settings have been saved (initially empty)
1387        lbl_status = tk.Label(root, text="", fg="green")
1388        lbl_status.grid(row=12, column=1, columnspan=2, pady=5)
```

Figure 79 New function for VVIR mode cont.

```
1390    # Handle submission validation, refer to VOOR function of same name
1391    def handle_submit():
1392        current_device = find_device()
1393        params = defaultParams()
1394
1395        try:
1396            lrl = int(entry_lrl.get())
1397            url = int(entry_url.get())
1398            msr = int(entry_msr.get())
1399            at = float(entry_at.get())
1400            rt = int(entry_rt.get())
1401            rf = int(entry_rf.get())
1402            rct = int(entry_rct.get())
1403            va = float(entry_va.get())
1404            va = round(va, 1)
1405            if (va>5 or va<0):
1406                raise ValueError("Amplitude must not be between 0-5.0")
1407            pw = int(entry_pw.get())
1408            pw = round(pw)
1409            if (pw < 1 or pw > 30):
1410                raise ValueError("Pulse Width must be between 1-30")
1411            vsens = int(entry_vs.get())
1412            if (vsens > 5 or vsens < 0):
1413                raise ValueError("Sensitivity must be between 0-5.0")
1414            vrp = int(entry_vrp.get())
1415
1416        except ValueError:
1417            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
1418            return
1419
1420        if current_device is None:
1421            lbl_status.config(text="Please connect a board!", fg="r")
1422            return
1423
1424        params["mode"] = 4
1425        params["rate_adapt"] = 1
1426        params["lrl"] = lrl
1427        params["url"] = url
1428        params["vent_amp"] = va
1429        params["vent_pw"] = pw
1430        params["vent_sens"] = vsens
1431        params["vrp"] = vrp
1432        params["msr"] = msr
1433        params["act_thresh"] = at
1434        params["reaction_time"] = rt
1435        params["response_fact"] = rf
1436        params["recovery_time"] = rct
1437
```

Figure 80 New function for VVIR mode cont.

```
1438        save_settings('VVIR', params)
1439        serialCom.send_parameters(params,currPort)
1440        lbl_status.config(text="Settings have been saved!", fg="green")
1441        root.after(3000, lambda: lbl_status.config(text=""))
1442
1443    # Submit button
1444    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
1445    btn_submit.grid(row=13, column=1, columnspan=2, pady=10)
1446
1447    # Back button to go to the mode_picker screen
1448    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
1449    btn_back.grid(row=15, column=1, columnspan=2, pady=(10, 20))
```

Figure 81 New function fro VVIR mode cont.

```python
# For AAIR parameters
def open_aair_pacing_settings():
    clear_window()
    root.title("AAIR Pacing Settings")
    root.geometry("400x550")
    root.grid_columnconfigure(0, weight=1)
    root.grid_columnconfigure(1, weight=0)
    root.grid_columnconfigure(2, weight=0)
    root.grid_columnconfigure(3, weight=1)
    root.grid_rowconfigure(0, weight=1)
    root.grid_rowconfigure(9, weight=1)

    lbl_title = tk.Label(root, text="AAIR Pacing Settings", font=("Helvetica", 14, "bold"))
    lbl_title.grid(row=0, column=1, columnspan=2, pady=(10, 20))

    # Lower Rate Limit (LRL)
    lbl_lrl = tk.Label(root, text="Lower Rate Limit (LRL):")
    lbl_lrl.grid(row=1, column=1, sticky="e", padx=10, pady=5)
    entry_lrl = tk.Entry(root)
    entry_lrl.grid(row=1, column=2, padx=10, pady=5)

    # Upper Rate Limit (URL)
    lbl_url = tk.Label(root, text="Upper Rate Limit (URL):")
    lbl_url.grid(row=2, column=1, sticky="e", padx=10, pady=5)
    entry_url = tk.Entry(root)
    entry_url.grid(row=2, column=2, padx=10, pady=5)

    #max sensor rate
    lbl_msr = tk.Label(root, text="Maximum Sensor Rate:")
    lbl_msr.grid(row=3, column=1, sticky="e", padx=10, pady=5)
    entry_msr = tk.Entry(root)
    entry_msr.grid(row=3, column=2, padx=10, pady=5)

    #activity threshold
    lbl_at = tk.Label(root, text="Activity Threshold:")
    lbl_at.grid(row=4, column=1, sticky="e", padx=10, pady=5)
    entry_at = tk.Entry(root)
    entry_at.grid(row=4, column=2, padx=10, pady=5)

    #reaction time
    lbl_rt = tk.Label(root, text="Reaction Time")
    lbl_rt.grid(row=5, column=1, sticky="e", padx=10, pady=5)
    entry_rt = tk.Entry(root)
    entry_rt.grid(row=5, column=2, padx=10, pady=5)

    #response factor
    lbl_rf = tk.Label(root, text="Response Factor")
    lbl_rf.grid(row=6, column=1, sticky="e", padx=10, pady=5)
    entry_rf = tk.Entry(root)
    entry_rf.grid(row=6, column=2, padx=10, pady=5)
```

Figure 82 New function for AAIR mode

```
1503        #recovery time
1504        lbl_rct = tk.Label(root, text="Recovery Time")
1505        lbl_rct.grid(row=7, column=1, sticky="e", padx=10, pady=5)
1506        entry_rct = tk.Entry(root)
1507        entry_rct.grid(row=7, column=2, padx=10, pady=5)
1508
1509        # Atrial Amplitude
1510        lbl_aa = tk.Label(root, text="Atrial Amplitude:")
1511        lbl_aa.grid(row=8, column=1, sticky="e", padx=10, pady=5)
1512        entry_aa = tk.Entry(root)
1513        entry_aa.grid(row=8, column=2, padx=10, pady=5)
1514
1515        # Atrial Pulse Width
1516        lbl_pw = tk.Label(root, text="Atrial Pulse Width:")
1517        lbl_pw.grid(row=9, column=1, sticky="e", padx=10, pady=5)
1518        entry_pw = tk.Entry(root)
1519        entry_pw.grid(row=9, column=2, padx=10, pady=5)
1520
1521        # Atrial Sensitivity
1522        lbl_as = tk.Label(root, text="Atrial Sensitivity:")
1523        lbl_as.grid(row=10, column=1, sticky="e", padx=10, pady=5)
1524        entry_as = tk.Entry(root)
1525        entry_as.grid(row=10, column=2, padx=10, pady=5)
1526
1527        # ARP
1528        lbl_arp = tk.Label(root, text="ARP:")
1529        lbl_arp.grid(row=11, column=1, sticky="e", padx=10, pady=5)
1530        entry_arp = tk.Entry(root)
1531        entry_arp.grid(row=11, column=2, padx=10, pady=5)
1532
1533        # PVARP
1534        lbl_pvarp = tk.Label(root, text="PVARP:")
1535        lbl_pvarp.grid(row=12, column=1, sticky="e", padx=10, pady=5)
1536        entry_pvarp = tk.Entry(root)
1537        entry_pvarp.grid(row=12, column=2, padx=10, pady=5)
1538
1539        lbl_status = tk.Label(root, text="", fg="green")
1540        lbl_status.grid(row=13, column=1, columnspan=2, pady=5)
```

Figure 83 New function for AAIR mode cont.

```
1543    def handle_submit():
1544        current_device = find_device()
1545        params = defaultParams()
1546
1547        try:
1548            lrl = int(entry_lrl.get())
1549            url = int(entry_url.get())
1550            msr = int(entry_msr.get())
1551            at = float(entry_at.get())
1552            rt = int(entry_rt.get())
1553            rf = int(entry_rf.get())
1554            rct = int(entry_rct.get())
1555            aa = float(entry_aa.get())
1556            pw = int(entry_pw.get())
1557            asens = int(entry_as.get())
1558            arp = int(entry_arp.get())
1559            pvarp = int(entry_pvarp.get())
1560
1561        except ValueError:
1562            lbl_status.config(text="Please enter all fields as valid numbers!", fg="r")
1563            return
1564        if current_device is None:
1565            lbl_status.config(text="Please connect a board!", fg="r")
1566            return
1567
1568        params["mode"] = 3
1569        params["rate_adapt"] = 1
1570        params["lrl"] = lrl
1571        params["url"] = url
1572        params["atr_amp"] = aa
1573        params["atr_pw"] = pw
1574        params["atr_sens"] = asens
1575        params["arp"] = arp
1576        params["pvarp"] = pvarp
1577        params["msr"] = msr
1578        params["act_thresh"] = at
1579        params["reaction_time"] = rt
1580        params["response_fact"] = rf
1581        params["recovery_time"] = rct
1582
1583        save_settings('AAIR', params)
1584        serialCom.send_parameters(params,currPort)
1585        lbl_status.config(text="Settings have been saved!", fg="green")
1586        root.after(3000, lambda: lbl_status.config(text=""))
1587
1588    # Submit button
1589    btn_submit = tk.Button(root, text="Submit", command=handle_submit)
1590    btn_submit.grid(row=14, column=1, columnspan=2, pady=10)
1591
1592    # Back button to go to the mode_picker screen
1593    btn_back = tk.Button(root, text="Back", command=mode_picker, bg="#E49B0F", fg="white", width=10, height=1)
1594    btn_back.grid(row=16, column=1, columnspan=2, pady=(10, 20))
```

Figure 84 New function for AAIR mode cont.

```
# Bind the Enter key to trigger the login button
root.bind('<Return>', on_login_enter_key)

def show_login_screen():
    clear_window()  # Clear the window

    # Recreate and display the login screen content
    root.title("Pacemaker DCM")
    root.geometry("500x400")
    root.configure(bg="#f0f0f0")

    # Welcome message
    welcome_message = tk.Label(root, text="\n\tWelcome\n", font=("Cambria", 24, "bold"), bg="#f0f0f0")
    welcome_message.grid(row=0, column=0, columnspan=3, pady=(20, 0))  # Center by columnspan

    welcome_message2 = tk.Label(root, text="\t              Please Login or Register Below", font=("Cambria", 12), bg="#f0f0f0")
    welcome_message2.grid(row=1, column=0, columnspan=3, pady=(0, 30))

    # Name label and entry (Global variables for later use)
    lbl_name = tk.Label(root, text="Name:", bg="#f0f0f0")
    lbl_name.grid(row=2, column=0, padx=(100, 0), pady=5)
    global entry_name
    entry_name = tk.Entry(root, width=30)
    entry_name.grid(row=2, column=1, padx=(0, 0), pady=5)

    # Password label and entry
    lbl_password = tk.Label(root, text="Password:", bg="#f0f0f0")
    lbl_password.grid(row=3, column=0, padx=(100, 0), pady=5)
    global entry_password
    entry_password = tk.Entry(root, show="*", width=30)
    entry_password.grid(row=3, column=1, padx=(0, 0), pady=5)

    # Register button (Global variables for later use)
    global btn_register
    btn_register = tk.Button(root, text="Register", command=register_user, bg="#4CAF50", fg="green", width=10, height=1)
    btn_register.grid(row=4, column=1, padx=(0, 0), pady=20, sticky="e")

    # Login button with green outline (Global variables for later use)
    global btn_login
    btn_login = tk.Button(root, text="Login", command=login_user, bg="white", fg="green", width=10, height=1)
    btn_login.grid(row=4, column=1, padx=(0, 0), pady=20, sticky="w")

    # Bind the Enter key to trigger the login button
    root.bind('<Return>', on_login_enter_key)
```

Figure 85 New code for logout button logic

The following new code below is found in the files globalVars.py and serialCom.py respectively.

```
def defaultParams():
    global params
    params = {
        "mode": 0,
        "rate_adapt": 0,
        "lrl": 60,
        "url": 60,
        "vent_amp": 1.0,
        "atr_amp": 1.0,
        "vent_pw": 20,
        "atr_pw": 20,
        "vent_sens": 4,
        "atr_sens": 4,
        "vrp": 20,
        "arp": 20,
        "pvarp": 1,
        "act_thresh": 0.5,
        "reaction_time": 30,
        "response_fact": 10,
        "recovery_time": 30,
        "msr": 120
    }
    return params
```

Figure 86 Default parameters dictionary code

```python
import serial
import serial.tools.list_ports
import struct

firstByte = 22
send = 13+6
echo = 22 + 6
get_egram = 47 + 6

# Send parameters to pacemaker, accepts parameters
def send_parameters(params,currPort):
    st = struct.Struct('<bbBBfBfBBBBBBBBBfB')
    send_data = st.pack(firstByte,send,params["mode"],params["atr_pw"],params["atr_amp"],
            params["vent_pw"],params["vent_amp"],params["lrl"],params["url"],
            params["arp"],params["vrp"],params["atr_sens"],params["vent_sens"],
            params["rate_adapt"],params["msr"],params["reaction_time"],params["recovery_time"],
            params["act_thresh"],params["response_fact"])
    with serial.Serial(currPort, baudrate=115200,timeout=1) as ser:
        ser.write(send_data)
        ser.close()

# Read sent parameters from pacemaker
def read_params(params,currPort):
    st = struct.Struct('<bbBBfBfBBBBBBBBBfB')
    send_data = st.pack(firstByte,echo,params["mode"],params["atr_pw"],params["atr_amp"],
            params["vent_pw"],params["vent_amp"],params["lrl"],params["url"],
            params["arp"],params["vrp"],params["atr_sens"],params["vent_sens"],
            params["rate_adapt"],params["msr"],params["reaction_time"],params["recovery_time"],
            params["act_thresh"],params["response_fact"])

    with serial.Serial(currPort, baudrate=115200,timeout=1) as ser:
        ser.write(send_data)
        curr_params = ser.read(42) # reading from pacemaker
        ser.close()
    return curr_params # returns list of current parameters

# begins and ends communication of egram data, accepts bool send_data
def get_plotData(ser,params):
    st = struct.Struct('<BBfBfBBBBBBBBBfB')
    send_data = b'\x16' + b'\x47' + st.pack(params["mode"],params["atr_pw"],params["atr_amp"],
            params["vent_pw"],params["vent_amp"],params["lrl"],params["url"],
            params["arp"],params["vrp"],params["atr_sens"],params["vent_sens"],
            params["rate_adapt"],params["msr"],params["reaction_time"],params["recovery_time"],
            params["act_thresh"],params["response_fact"])
    ser.write(send_data)
    data = ser.read(32)
    st_read = struct.Struct('<dd')
    unpacked = st_read.unpack(data)

    try:
        voltage = unpacked
    except IndexError:
        print("No data found")
        return None
    return voltage
```

Figure 87 New code for serial communication

Validation and Verification

Simulink (Assignment 1)

Validation:

We were required to implement 4 pacing modes, AOO, VOO, AAI, and VII. All 4 pacing modes are included in the pacemaker. We were required to implement the following parameters: LRL, URL, separate pulse width for ventricle and atrial, separate amplitude for ventricle and atrial, ARP, VRP, atrial sensitivity, and ventricle sensitivity. All of the variables are implemented into the pacemaker design.

Verification:

The 4 pacing modes required were tested using heartview and are working. For each of the following tests, the heart rate was set to 60BPM and a pulse width of 20ms.

AOO: Pulses are sent to the atrial whether or not there are incoming beats from the heart:

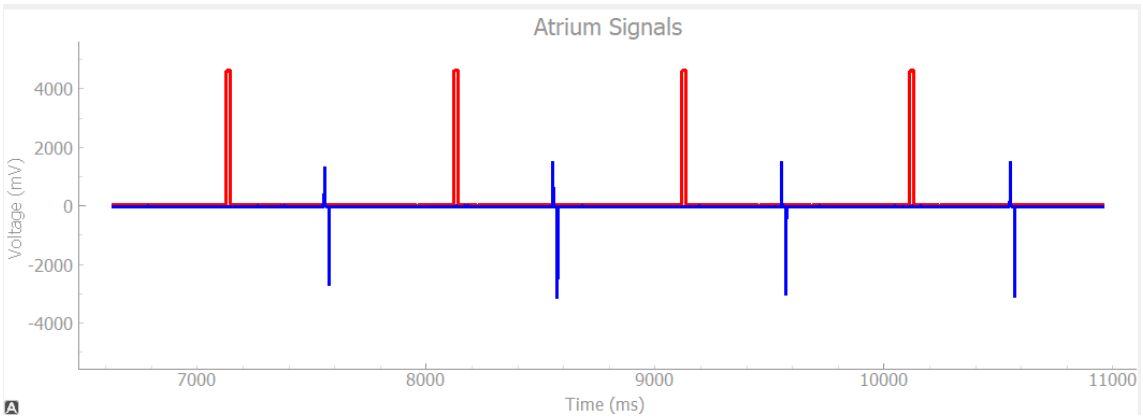


Figure 88: AOO Heartview Graph

VOO: Pulses are sent to the ventricle whether or not there are incoming beats from the heart:

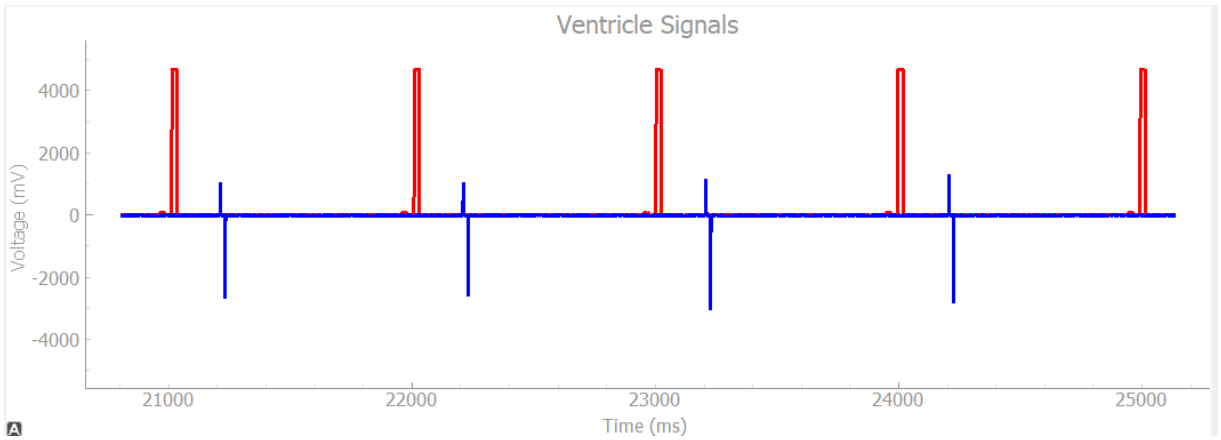


Figure 89: VOO Heartview Graph

AAI: Pulses are sent to the atrial but stop when there are incoming beats from the heart. They also begin to pulse when the beats stop:

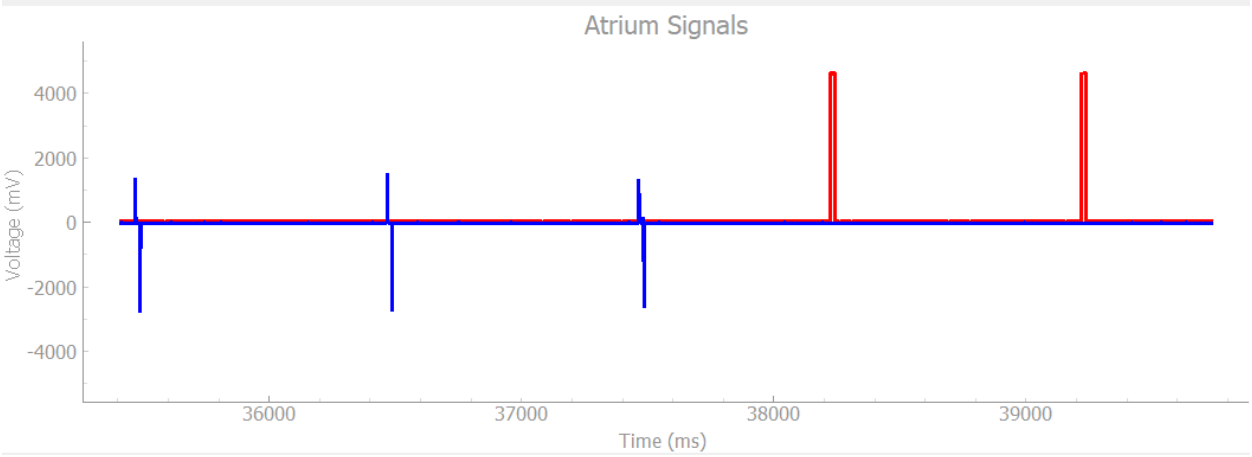


Figure 90: AAI Heartview Graph 1

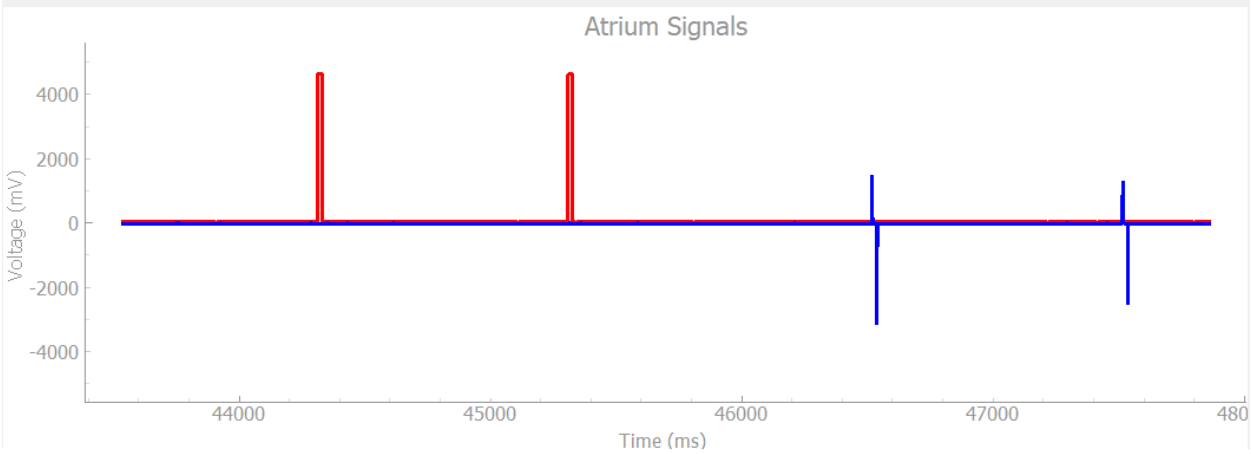


Figure 91: AAI Heartview Graph 2

VVI: Pulses are sent to the ventricle but stop when there are incoming beats from the heart. They also begin to pulse when the beats stop:

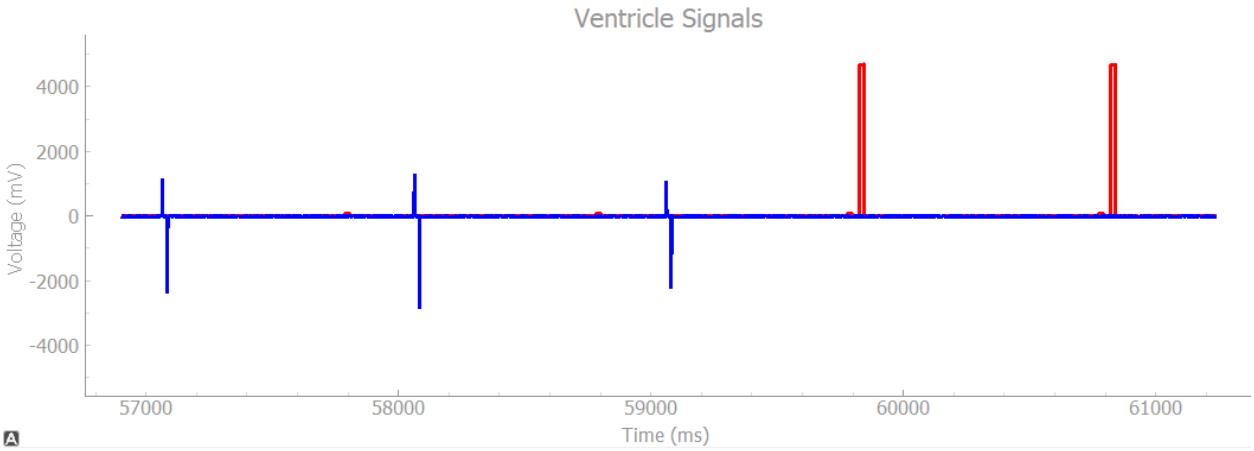


Figure 92: VVI Heartview Graph 1

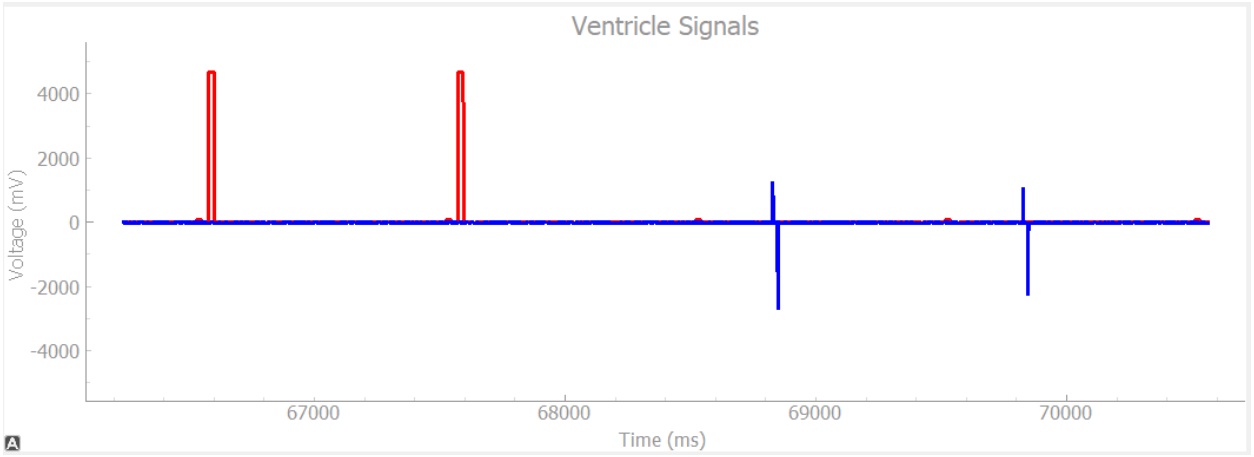


Figure 93: VVI Heartview Graph 2

Simulink (Assignment 2)

Validation:

We were required to implement the rate adaptive modes AOOR, VOOR, AAIR, and VVIR. We were also required to implement the serial communication between Simulink and the DCM, which is also used for the EGRAM data. The double modes were also required as a bonus, and when the button is pressed the ventricular mode would be turned off but the atrial would stay on. All these requirements were implemented into the Simulink model.

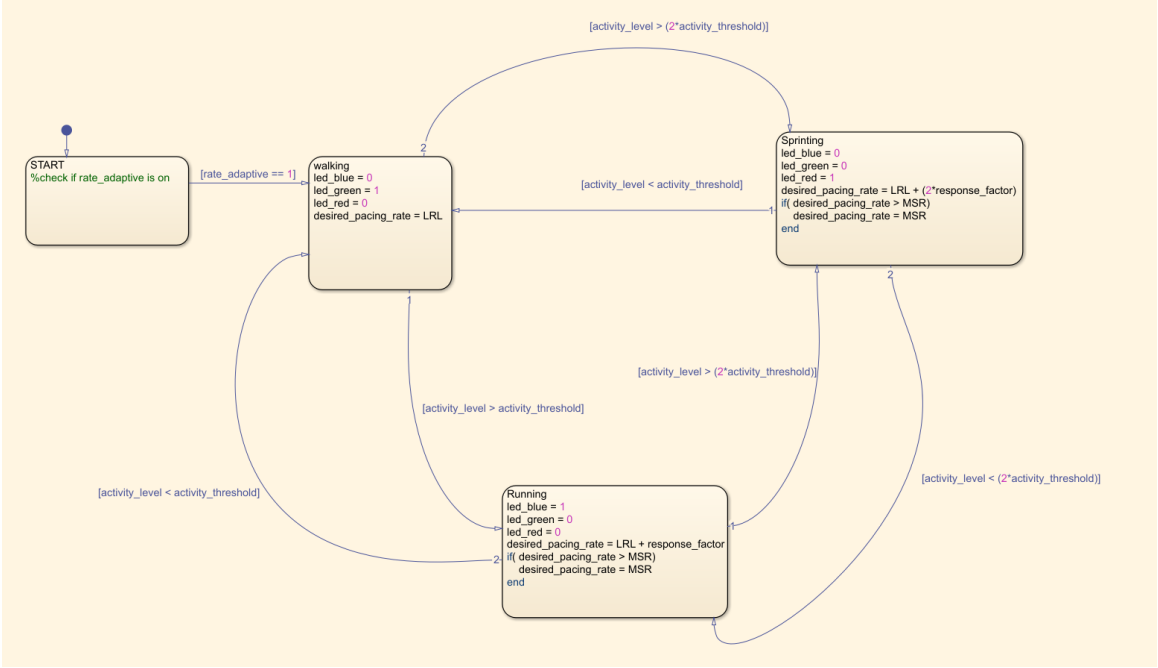


Figure 94: Rate Adaptive Implementation

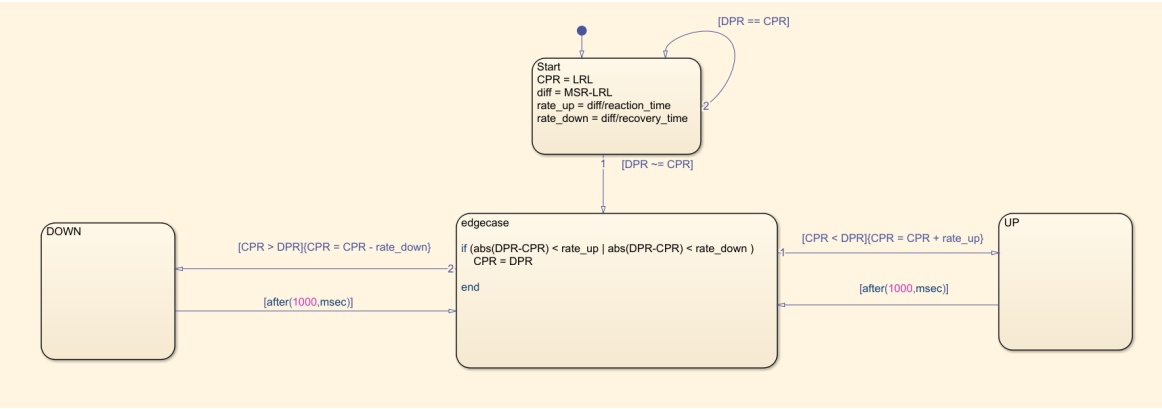


Figure 95: Rate Adaptive Implementation

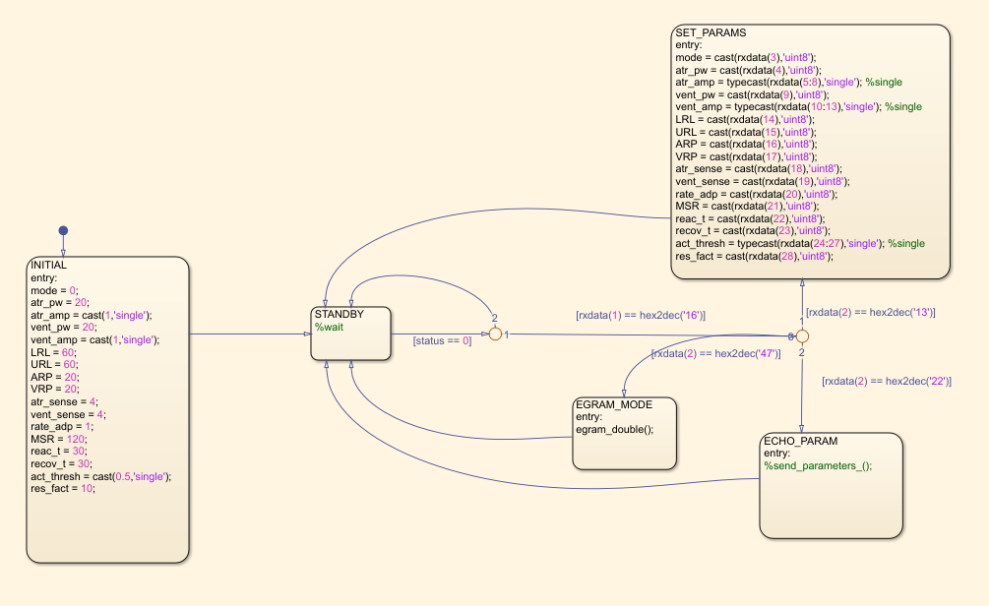


Figure 96: Serial Implementation

Verification:

The rate adaptive modes increase to the desired pacing rate of the after being shaken. In the walking mode the LED turns green, in the running mode the LED turns blue, in the sprinting mode the LED turns red.

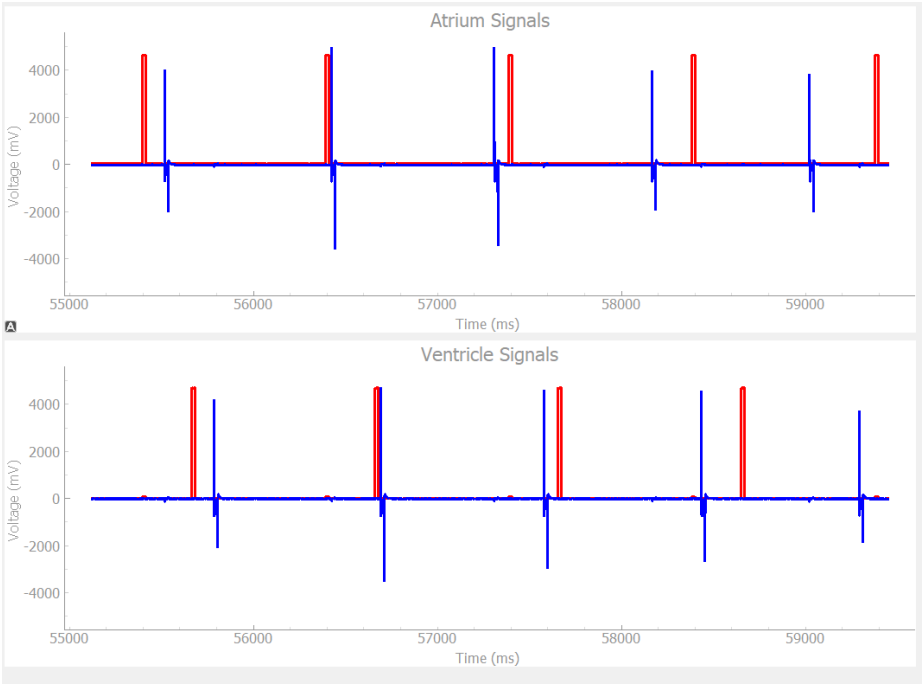


Figure 97: Rate Adaptive Increasing

When shaken, the rate adaptive mode increases the current pacing rate and the time between pulses gets smaller.

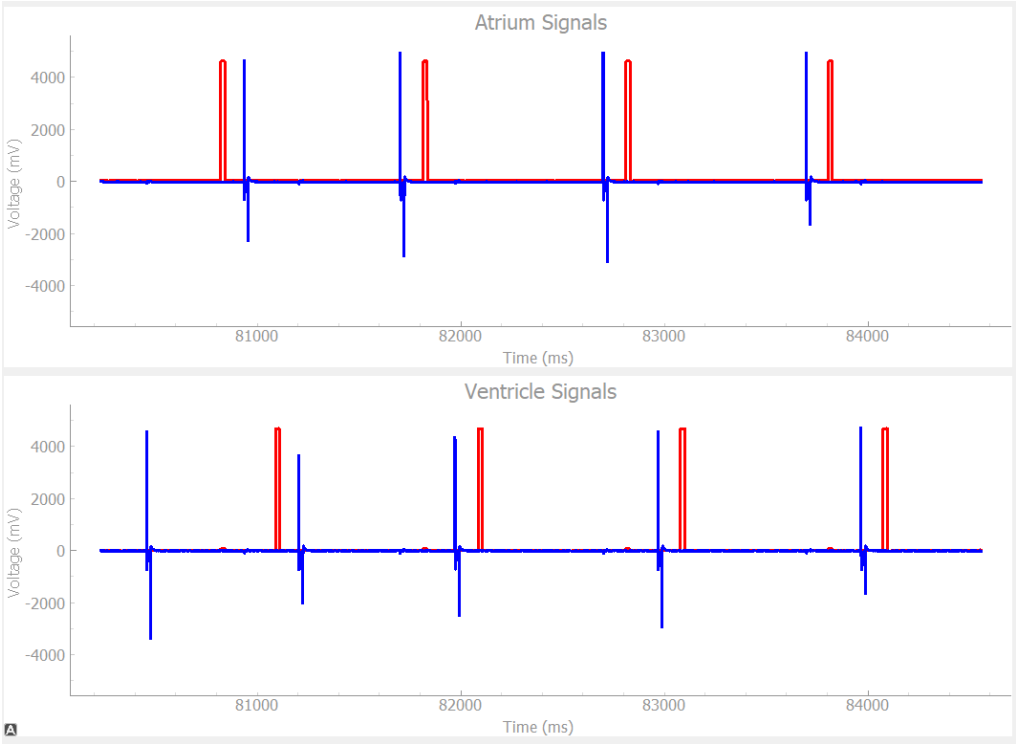


Figure 98: Rate Adaptive Decreasing

After being shaken and then stopped, the current pacing rate decreases and the time between pulses gets longer.

The serial logic sends and receives all necessary parameters in the correct type, using byte packing when necessary for larger data types. The EGRAM logic is also fully implemented and working.

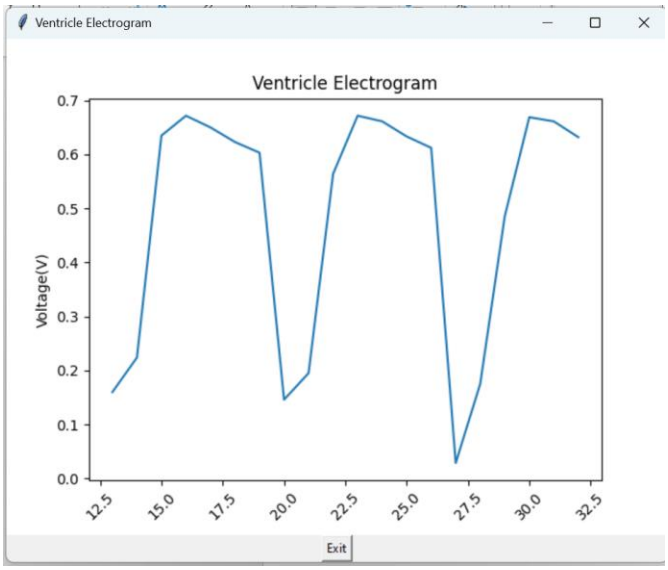


Figure 99: EGRAM

The double modes also work and using the push button on the pacemaker inhibits the ventricular pacing, while the atrial pacing continues.

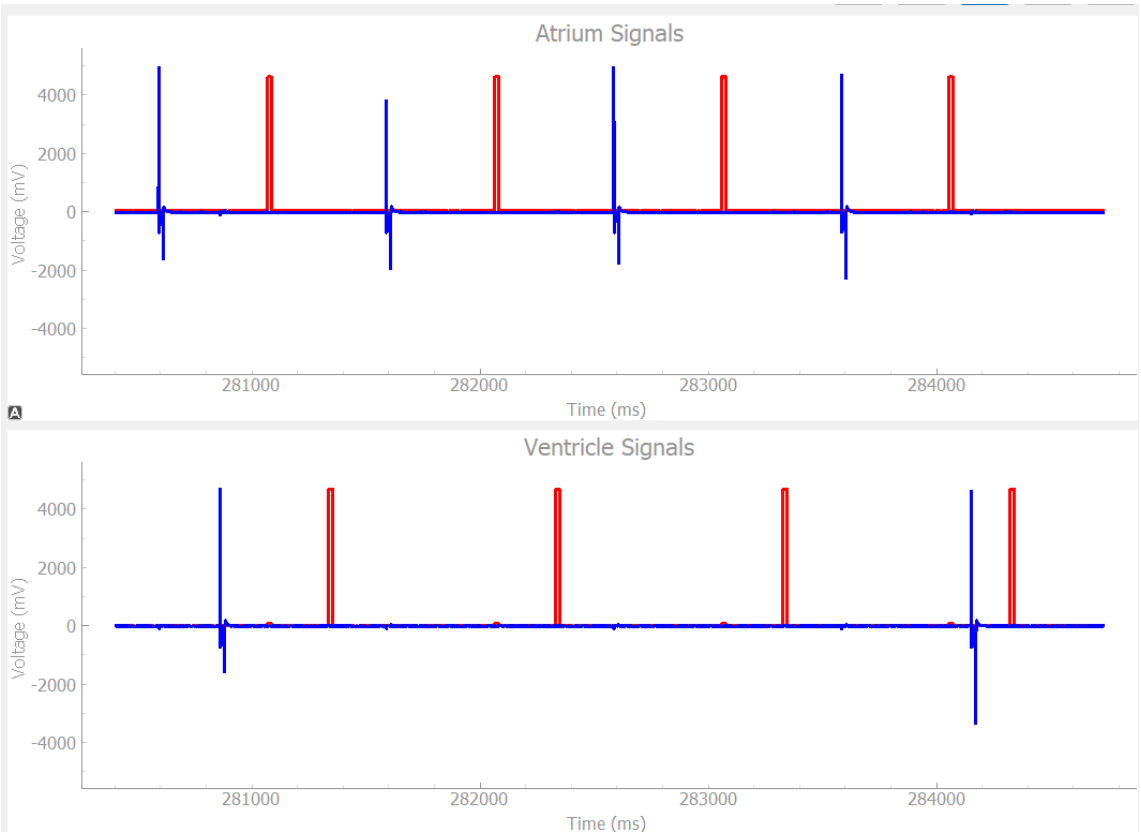


Figure 100: DOOR with Push Button Ventricle Inhibiting

DCM (Assignment 1)

The DCM was required to first, have a welcome screen where users are able to login or register a new user, with the constraint that only 10 users may be stored locally. The welcome screen window created is displayed in the figure below. A user will successfully login if they input the correct username and password that is already saved, or if they register a new user that has not yet been created and if 10 logins have not already been saved.

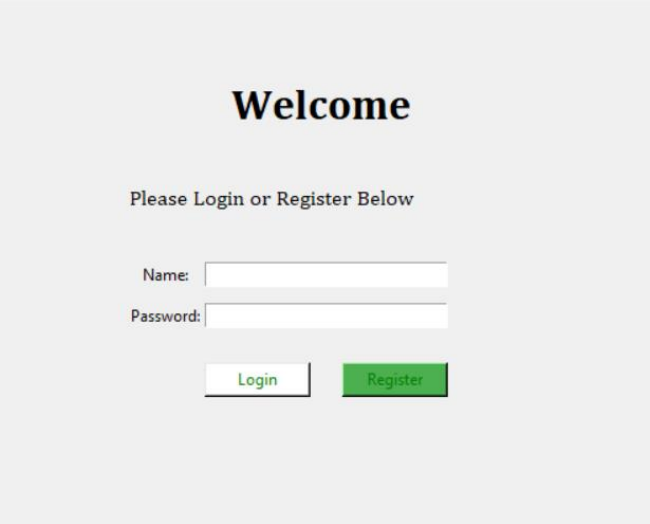


Figure 101 Welcome/Login page

If the login is unsuccessful or a pre-existing or 11th user is trying to register, an error message will pop up detailing the user could not be admitted. Once a correct username and password is provided, the user will see a welcome message and will have access to the rest of the interface.

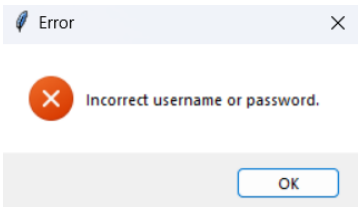


Figure 102 Login error message

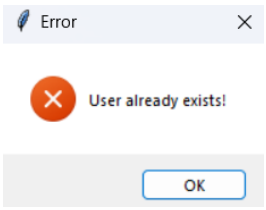


Figure 103 Registration error message

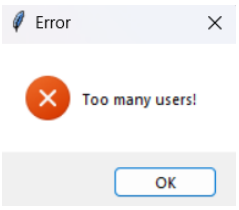


Figure 104 Registration error message

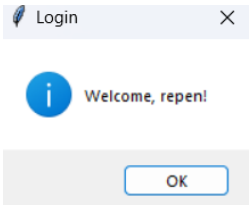


Figure 105 Successful login message

Next, the DCM was required to display all programmable parameters for review and modification as well as present all the pacing modes AOO, VOO, AAI, and VVI. After selecting a mode from the window displayed below Figure 48, the user is able to enter in the values to the applicable parameters, shown in Figures 49-52.

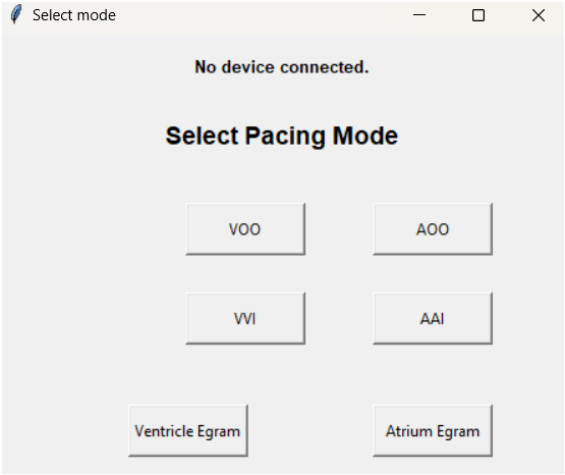


Figure 106 Pacing selection window

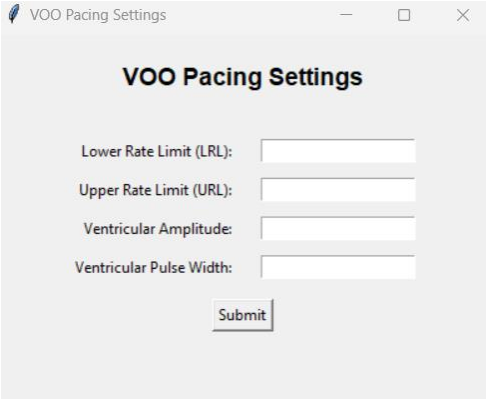


Figure 107 VOO pacing settings window

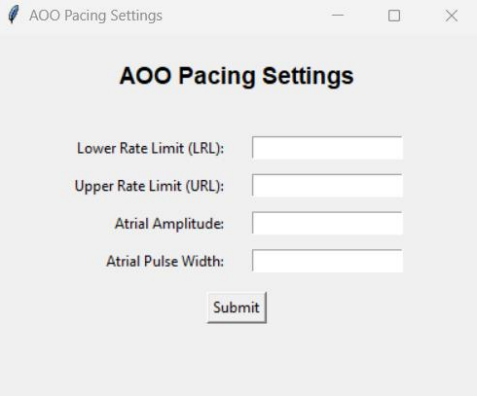


Figure 108 AOO pacing settings window

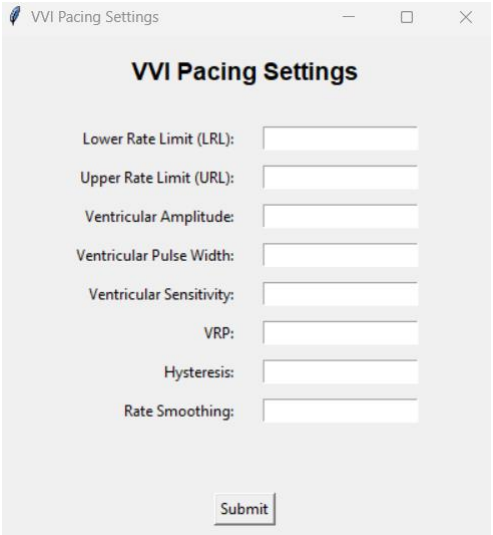


Figure 109 VVI pacing settings window

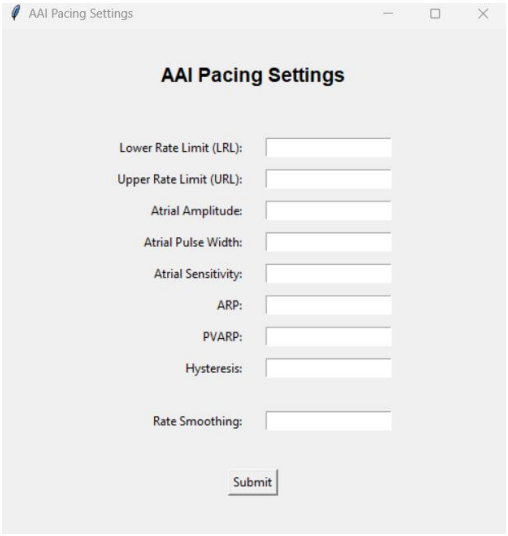


Figure 110 AII pacing settings window

Another requirement is for the interface to be capable of indicating when the DCM and device are communicating as well as indicate if a different pacemaker device is connected

than what was connected previously. This interface informs the user if and what device is connected by displaying a message at the top of the pacing selection window. As seen in Figure 48, the message at the top of the page reads “No device connected” informing the user no device is currently sensed. When a device is connected the message will change to “Now communicating with device: X”, where X is the serial number of the device. If a new device is connected a message will pop up detailing that the device has not yet been connected and will save the serial number to a csv file. The message will at the top of the mode selection window will also change and display the serial number of the new device.

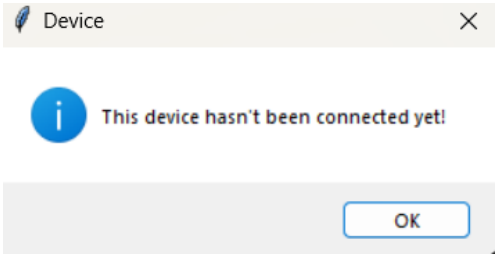


Figure 111 New device message

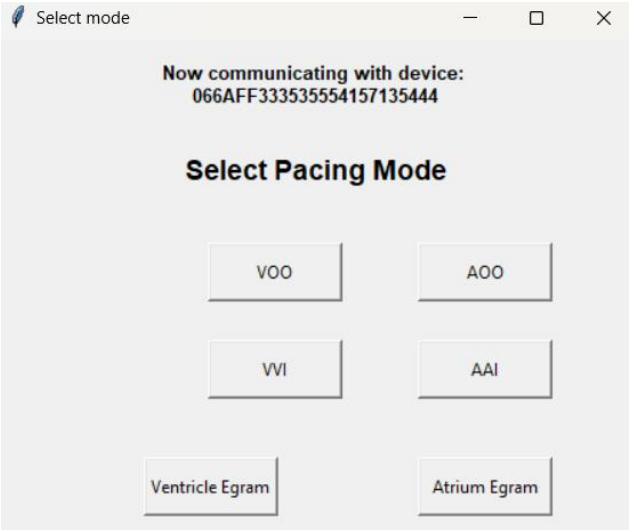


Figure 112 Mode selection page featuring serial number of connected device

This DCM also implements two data structures to handle egram data, as stated in the requirements. The code for the data structures can be seen above in Figures 10-12. These are being implemented currently with randomly generated data and the plots can be seen when the user selects either the Ventrical Egram or Atrium Egram when on the mode selection window (Figure 48). The following animated plots will pop up when selected. In future assignments we will be able to generate plots utilizing actual values instead of randomly generated data.

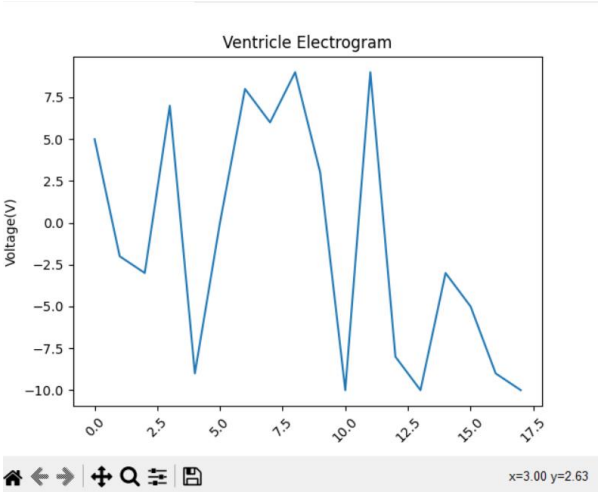


Figure 113 Generated plot of ventricle electrogram

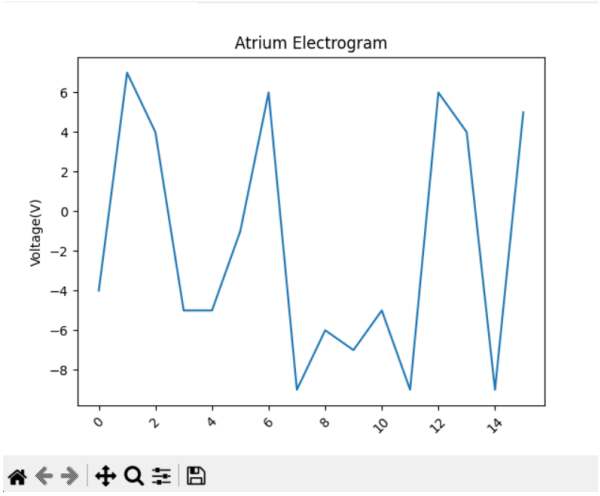


Figure 114 Generated plot of atrium electrogram

One of the safety measures taken is the requirement of a device connection in order to successfully input settings for the pacing modes. Additionally, if the inputs are invalid floats (if they are left blank or are characters), the user will receive an error message and will have to try again. Only when there is a device connected and all of the inputs are filled in and are valid floats will the settings be saved successfully, and the user will receive a confirmation message.

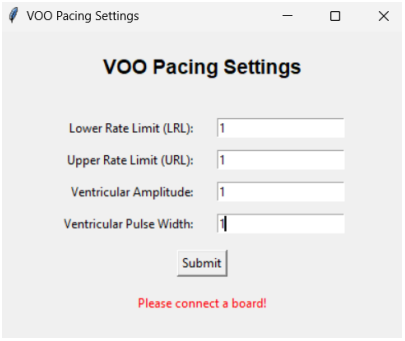


Figure 115 No device connected error message

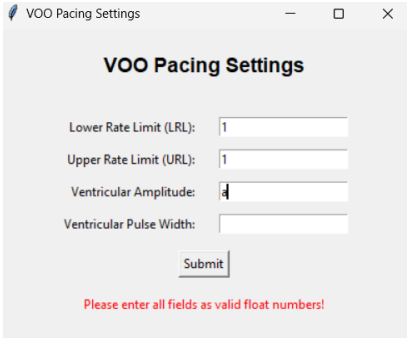


Figure 116 Invalid inputs error connection

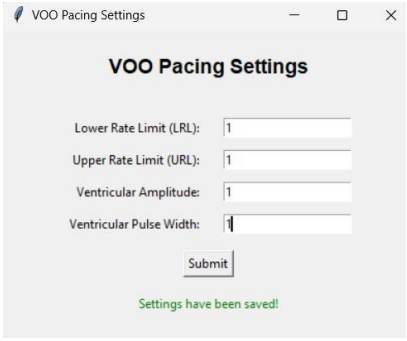


Figure 117 Successful submission message

Additional features of the DCM user interface include the highlighted buttons at the login screen to login or register (seen in Figure 43), as well as the highlighted messages displayed after a user has submitted inputs for the modes (seen above in Figures 57-59). This can help enhance accessibility by making the options more noticeable and by emphasizing if the submission was successful or not. As well, the elements on the select mode window are adaptable to fit the size of the page whether the window is minimized or maximized or somewhere in between. As displayed in the figures below, the buttons adjust their spacing so that they are visible when the size of the window changes.

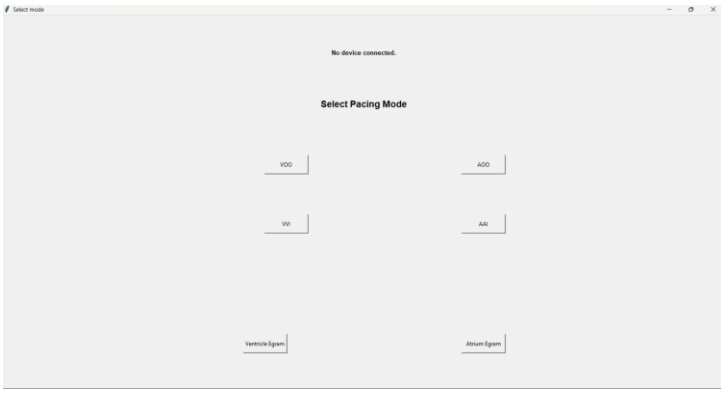


Figure 118 Button spacing when window is maximized

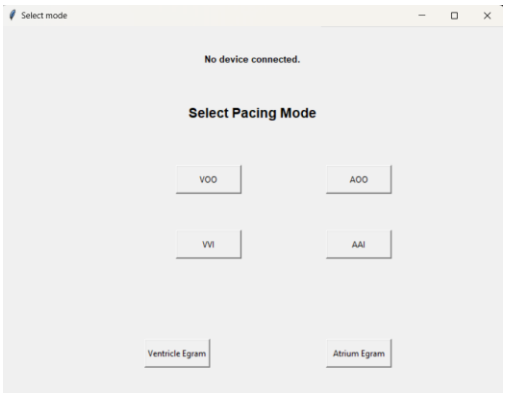


Figure 119 Button spacing when window is minimized

DCM (Assignment 2)

For Assignment 2, additional requirements have been outlined. The first being expanding the DCM to include the new pacing modes and their additional parameters. As shown in the Figure below, the new rate adaptive pacing modes have been added to the homepage.

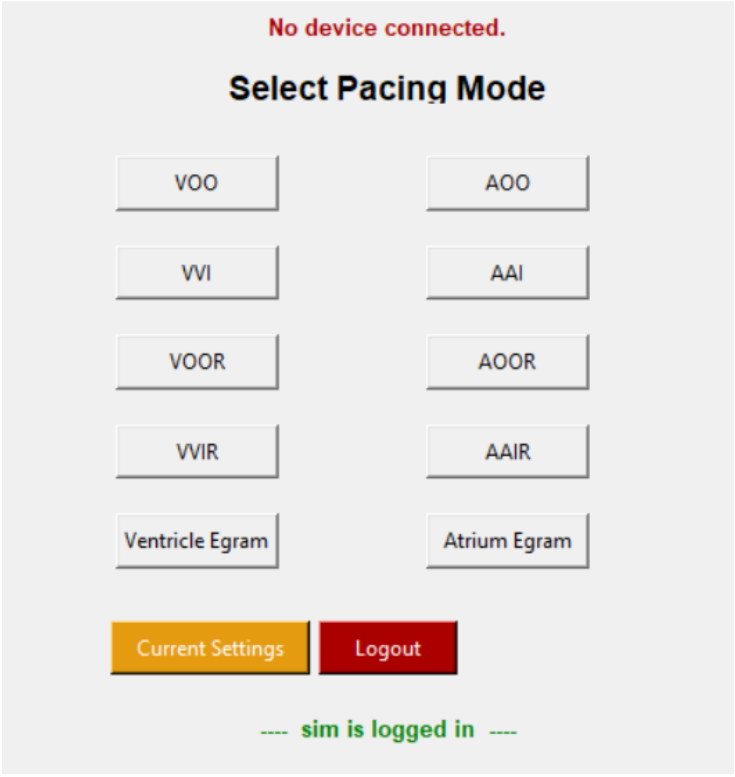


Figure 120 Updated select pacing mode homepage

When the new modes are selected, the pacing settings page includes the new parameters available to be inputted (activity threshold, response time, recovery time, maximum sensor rate, and response factor).

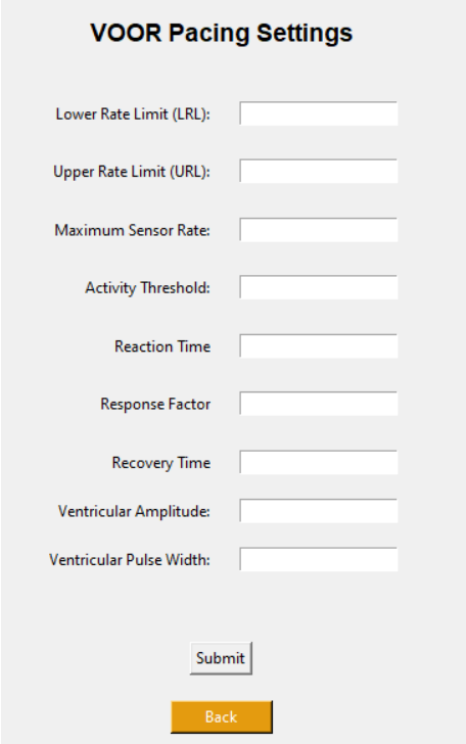


Figure 121 VOOR pacing settings

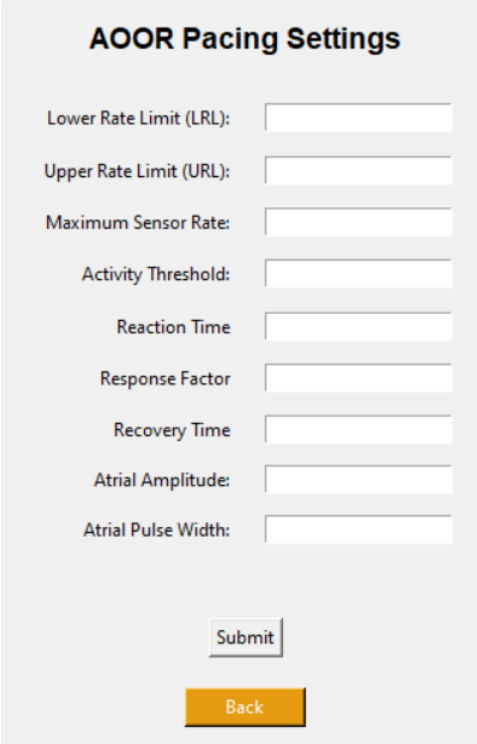


Figure 122 AOOR pacing settings

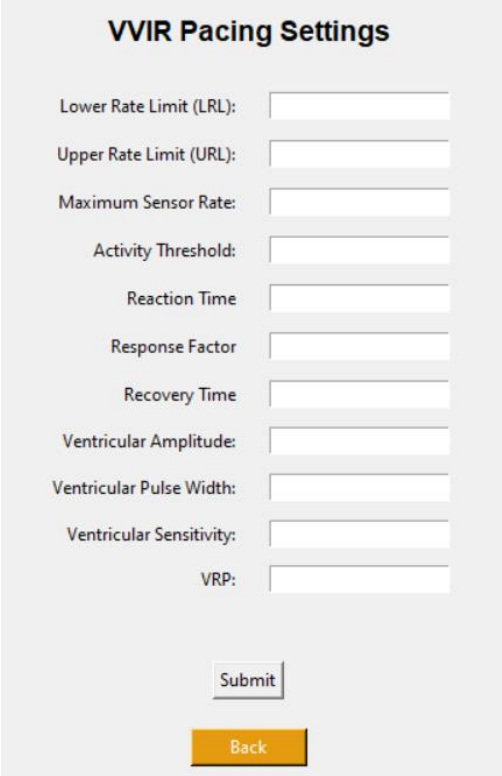


Figure 123 VVIR pacing settings

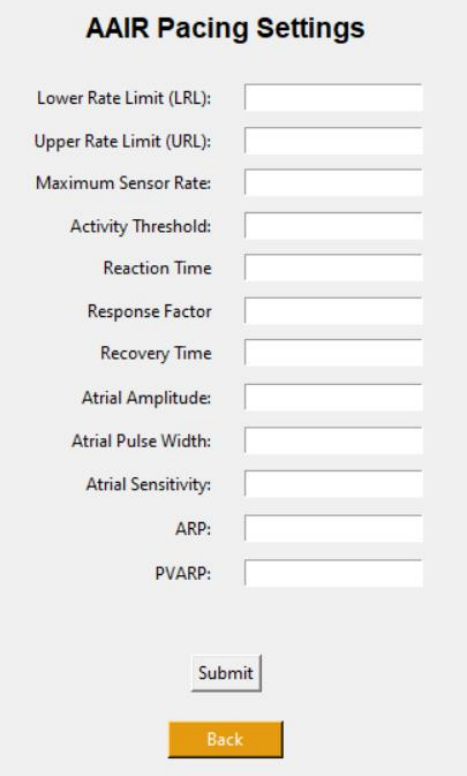


Figure 124 AAIR pacing settings

The next new requirement is to implement serial communication between the DCM and the pacemaker. This was done by utilizing the PySerial library. The DCM is able to communicate

the parameters users set by first converting them to bytes, and then sending writing them to the pacemaker device. The DCM was also able to receive data from the pacemaker by using the read() function. This is implemented when displaying the ventricle and atrium egram data, another one of the updated requirements. This was used as one way to confirm that our method of communicating with the pacemaker is successful is by looking at the plots generated. The egram graphs are displayed below.

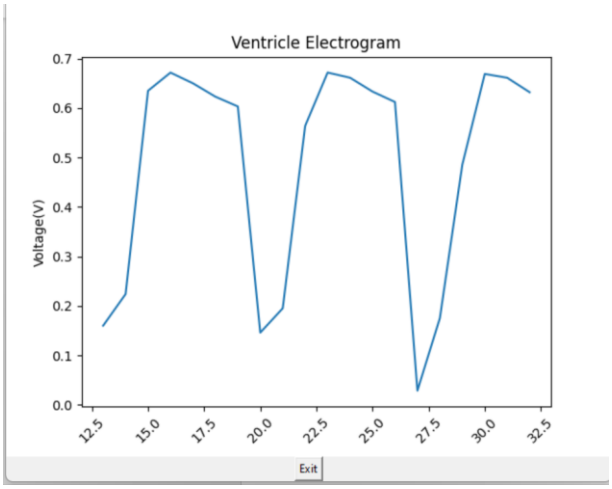


Figure 125 Ventricle egram plot

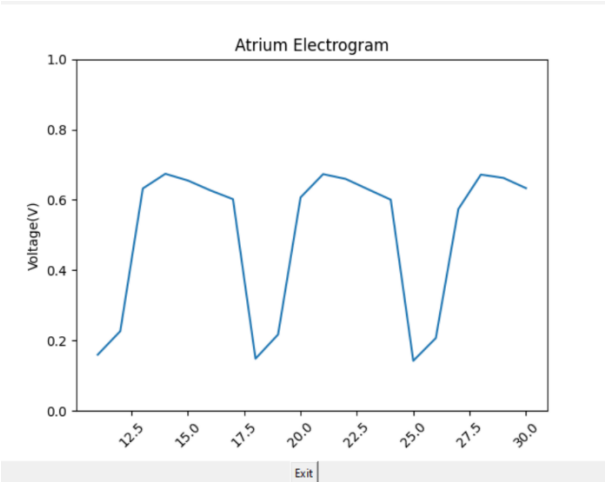


Figure 126 Atrium egram plot

Another additional requirement was to implement changes to the accepted programmable values for the atrial and ventricle amplitudes, pule widths, and sensitivities. Additionally, the accepted ranges for the new settings also needed to be implemented. The specific values for the accepted ranges for each setting are shown in Table 1, and the updated and new programmable parameters are shown in Table 2. The figure below displays the error message received when the inputs entered are not within the required range.

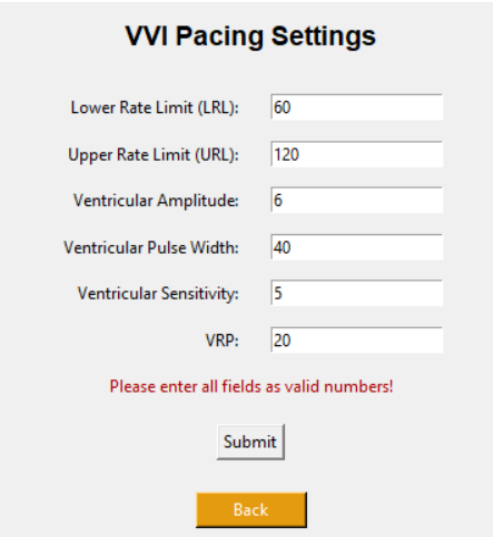


Figure 127 Error message for invalid entries

Part 2

Simulink (Assignment 1)

Requirements Changes

In the future there will be more pacemaker modes that will be required to be implemented. These include modes with rate modulation such as VOOR, VVIR, AOOR, AAIR. This will change our requirements by forcing us to consider more modes and change our logic to integrate with the DCM. This means that the DCM must be able communicate with the Simulink code and change parameters various. Modes with dual activity may also need to be implemented eventually. This changes the requirements for running multiple states and logic at the same time.

Design Decisions Changes

For rate adaptive modes, the pacemaker must be able to change the timing that it paces based on how often it senses pulses. This requires a change in how the sensing works. There is a period of time after a sense where the pacemaker is supposed to not check for a pulse in order to not double pace. However, if the pulse timing is changing then this delay must not exist, as it has to constantly be able to adapt to changing pulse rates.

Communicating with the DCM means that the parameters, which are currently just constants, need to be able to be changed through the DCM. This involves figuring out a way to send information from python to Simulink.

The modes with dual activity should have pretty simple design changes, as they would just run both sensing and pacing modes at the same time, however there may be safety issues with this concerning certain grounding pins.

Simulink (Assignment 2)

Implementing Previous Changes

As the group progressed through assignment two, the key requirement and design decision changes had to do with integrating rate adaptive pacing modes into the system. With assignment two came the addition of five addition pacing modes including AOOR, VOOR, AAIR, VVIR, and DOOR.

These requirement changes brought with them some major design changes as well. Rate adaptive pacing required a difference cycle for pacing and sensing which had to be implemented. Previously, there existed a delay after any heart activity in order to not double a pace where no pacing or sensing was done. With the new rate adaptive modes, pacing must be done much more consistently and therefore this delay had to be removed. All in all, the general program to do with difference pacing modes stayed true to the design from assignment one, focusing on modularity and low coupling between different modules of the code.

Additional things that needed to be added to the Simulink include serial communication, and an Electrogram (Egram) to display visual signals from the heart as a part of the user interface. In terms of program design, these required minimal changes to previous components due to the modularity of the design and were integrated with the rest of the program.

Requirement Changes

The program currently works with nine modes including different levels of pacing, sensing, inhibiting, and rate modulation. As a final integration to the program a fully functioning pacemaker would do well to have an effectively working DDDR mode. This will require a slight shift in program functionality and require some new pathways and logic within the code.

Design Decision Changes

Although a new mode would add some needed modules and/or logic, the modularity and low coupling of the design of the program would stay true to its current state. As things were added and changed the group would need to ensure the code stays readable. This involves hiding the logic not relevant for understanding and ensuring the code stays modular and clean as it progresses and scales.

Also as a future change, to make the pacing and overall function of the pacemaker, changes to the overall procedure of the program could be changed to account for hysteresis pacing. Hysteresis pacing reduces the need for constant pacing and can allow the pacemaker to function more efficiently in terms of power. This feature could another level of function to the system and overall provide a more complete experience/usage.

DCM (Assignment 1)

Requirements Changes

Moving forward with the project, some DCM requirements will likely have to change in response to the needs laid out in future assignments regarding the pacemaker. Particularly, the first likely to change is requirement 2, which relates to developing the essential functions of the pacemaker. As of now, only elements 1,2,3,4, and 7 of 3.2.2 in the PACEMAKER document were developed, but moving forward it will become increasingly important to ensure that all communications between the DCM and the device are indicated clearly. This means that it will become necessary to include more telemetry indicators (5 and 6), which would represent a change in the requirements of this assignment. Another requirement that is likely to change is requirement 3, which describes the necessary modes considered in the DCM. As of now only four pacing modes (VOO, AOO, VVI, AAI) were implemented in the DCM, but it is known that moving forward that these modes will need to be expanded upon to include a category to account for rate modulation (VOOR, AOOR, AAIR, VVIR). This would represent a departure from the current requirements, as an extra set of modes would have to be implemented in the final product. Requirement 5 is also likely to evolve, because now that the DCM and the Pacemaker will need to communicate, the data structures created to manage egram data will have to be applied to real-time data rather than used on the idealized random data currently. More requirements would likely also be affected because of the connection between the device and the DCM. Particularly, implementing the final concepts of user safety to separate data sent to the pacemaker between different logged in users would be an important requirement to focus on as the DCM begins to handle actual device data. All these requirements are likely to change and evolve as more information is given in future assignments.

Design Decisions Changes

Moving forward, accounting for both the weak areas in the current design iteration, and the requirement changes that are likely to happen, means that the design decisions made will have to change as well. First, the addition of more device modes would provide an opportunity to redesign the user interface (UI) to be more intuitive and well-created. This would likely take the form of the addition of elements such as: a logout button, a sidebar, a navbar, and a redesign of the home page to better guide the user and show the graphs of the Egram data with a much easier and intuitive UI.

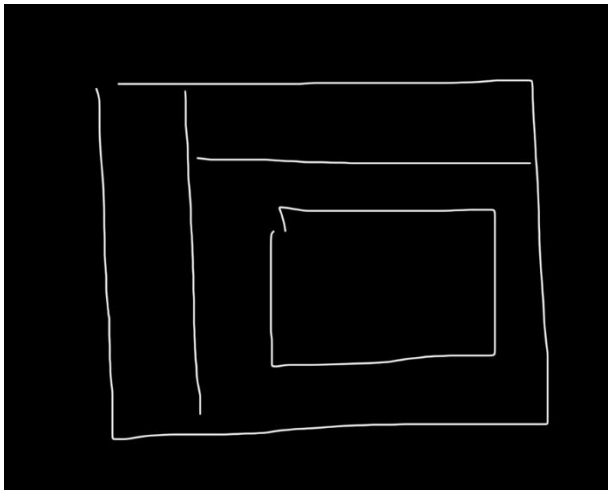


Figure 128: Rough sketch of likely future homepage, noting the addition of various UI elements

Another design decision likely to change would be the way that the DCM indicates to the user that the device and the program are communicating. Now that the two will have to transmit real data going forward, adding some kind of visual indicator to tell exactly when data is transferred between the two (like when user input is submitted) is a design decision worth looking into and will likely change in the coming assignments. Also, the design decisions made about separating different user's data would likely have to evolve to implement the layer of user safety mentioned above. As of now, several CSVs are being used to store the data needed for the program in the interim, but meeting the new requirements will present the opportunity to redesign the data storage system to better modulate users. This might take the form of a more robust file system, cloud storage, or another system chosen by the group. On top of this, the decisions made about plotting the Egram data will have to change as well. Implementing the ability to take real-time data and plot it will require different back-end infrastructure to handle the reception and visualization of the data. Finally, to tidy up the UI, a better way to manage the program's windows is a likely design decision that will be changed. Making sure that when a different menu is prompted, the previous one will close, and a robust way to navigate between windows is implemented will help improve performance, safety, and user experience by not allowing unexpected state switches when navigating through the program. Using the predicted requirements above allows for predictions to be made about what design decisions will be changed as well. Doing this, all the design decisions listed above are ones that are believed to be likely to change moving forward.

DCM (Assignment 2)

Implementing Previous Requirement Changes

It was important to remember requirement changes identified in assignment 1 and implement these changes in the DCM design created for assignment 2. Overall, the previously identified changes were implemented and contributed to a much more complete software.

The first requirement implemented was the addition and refinement of telemetry indicators to indicate communication between the DCM and the device. Before, it wasn't possible to indicate this communication because serial communication wasn't required in assignment 1. But with serial, steps were taken to ensure there are prompts that now clearly indicate communication. The most important prompt now occurs when settings are submitted to the device and notifies the user that these settings have been saved [Figure 70]. Before, this indicator was just indicative of the settings being written to a placeholder CSV, but now it has been tied to the execution of the serial communication logic, providing assurance that data is being transmitted when this message is seen. Also, indicators remain to tell users if there is a device connected and what that device is on the home screen in accordance with the requirements laid out.

The next requirement change implemented was the addition of the rest of the necessary modes for our pacemaker to function fully. Previously in assignment 1, no modes that involved rate adaptive pacing were needed, but this assignment has necessitated the creation of 4 more modes to handle rate adaptive pacing scenarios. Changing the number of possible modes from 4 to 8 and making them all fully functional by using previous mode display code as a template and expanding to include all necessary settings was an important step in ensuring that the product met the needs of its future hypothetical use cases.

Another important requirement change implemented was the plotting of real-time Egram data from the boards. Previously, no real data was being plotted because of the lack of serial communication, but with its implementation, steps were taken to pack and send the data as bytes through UART serial communication and the Serial module in Python to the DCM from Simulink using various blocks, and then to receive this data and plot it in real time. This was successful, and a demonstration of this fact was shown in demo.

Finally, requirements related to user safety and UI improvements were also met. Before, the submitted settings would all be saved to one CSV, which represented a user safety issue due to the lack of separation of data. To combat this, a file system that dynamically creates and removes CSV files for the users currently in the database was implemented

using Python and OS modules. This means that the data saved would be stored separately based on the current logged-in user and the system would handle the creation and deletion of files for new users automatically, lowering the risk of interference by a user and allowing better access to specific data. Also, a window navigation system was implemented, which allows users to move between each window without opening new windows each time. This makes it safer for the user to operate by preventing unexpected state changes by performing an operation in one window while another one is open.

All the above steps represent changes made to meaningfully improve our second design iteration while considering the requirement changes that were likely to occur after the first assignment.

Future Requirement Changes

Moving forward, there are changes to the requirements that are likely to occur to further improve the current DCM design. The first and most outstanding one would be the further refinement of the assignment 2 bonus and assuring that the DDDR mode works fully as necessitated in the assignment. As of now, the DOOR mode works completely as intended, and the DDDR mode is mostly functional, but is not well enshrined into the DCM and user interface. This means that requirement 3, which involved presenting all pacing modes, must be changed and expanded to fully include the D series modes.

Another requirement that is likely to change is the user interface requirement that determines how the user can navigate the DCM. As of now, the UI is designed to function well on a regular sized Windows/Mac/Linux, but it might be of future interest to expand to operating systems or displays beyond the typical PC. This would mean that the user interface requirements would change to match the needs of the specific port desired.

The next requirement which could see change in the future would be the serial communication or requirement 6, and how it packs and sends data to the DCM. Similarly in line with above, the serial communication protocol was designed for use with major PC operating systems, but if it were necessary to expand to other classes of devices, the communication protocol or steps taken would need to change to match the most efficient method for the specific device.

Finally, a last requirement that is likely to see change is the programmable parameters that are required to be implemented in the DCM. As of now, some parameters are not included because their corresponding modes weren't required to be implemented, and they provide

no direct impact on the Simulink handling of pacing. Moving forward however, as different pacing modes and situations could be considered, it seems likely that the pacing parameters would change to match the changes seen upstream.

Examining the total number of requirements likely to change overall, it's worth noting that the sheer number is much less than the first assignment. This is because the DCM approached a much more polished final form in this assignment compared to the last, which means that the predicted changes are often based on imagining ways the software could be used, rather than clear cut direction laid out in the assignment because there is not another one.

Design Decision Changes

When considering how the current DCM design could be brought in line with the predicted future requirement changes, it is clear various design decisions will need to evolve to make this possible.

The first decision likely to change would be the display of the different modes after the addition of DDDR and DOOR modes. If we were to keep the current styling for mode buttons, we would need to implement 10 total buttons in our mode picker screen which would likely be crowded and generally user unfriendly. Instead, it might make sense to implement a navbar with a dropdown menu for XOO, XXI, XXIR, and DXXR modes to both save space and provide a clearer user interface for the pacemakers. This is a design decision that would be worth investigating in the future when designing the DCM.

Another aspect of the UI which could change would be the visual display based on the type of device the DCM software is operating on. It would make sense to have different window layouts based on display dimension cutoffs that correspond to common devices that the software could be operated on. This would ensure that the UI remains usable and pleasant for the user across different platforms.

A more internal feature of the software might also see change, as the serial communication system might need to evolve to communicate in different ways to a more universal range of devices. This could potentially lead to the development of different data structures inside of the serial communication logic to abstract out commonalities across the different communication methods and allow for effective application of serial communication by taking steps such as creating data types that could package the EGRAM and settings data and send it through a variety of serial protocols.

Another design decision that could change is the implementation of different data entry fields in the form of a slider or arrow navigation field for certain parameters to ensure that the acceptable range and interval of change is clear. As of now, all imposed restrictions on the parameter range and interval are included, but because the data is required to be entered through keyboard input, it is not clear to the user what the range and interval is. In an effort to be more user friendly and transparent as to what is an acceptable parameter value, it might be worth considering different data entry fields so that the user is aware of the limitations on certain parameters which were imposed for safety.

Finally, a last design decision that could see change in the future is the implementation of default or initial values that are visible to the user. Currently, there are default values that are initialized when the program is run, but they are internal and hidden to the user, which makes it difficult to understand the baseline operating conditions of the pacemaker. Moving forward, printing these values inside of their corresponding fields when initially opening a mode might be beneficial to the user so they understand how the device and software responds nominally.

This is a collection of various design decisions that could foreseeably change in future iterations of the design. It is once again important to note that most of these changes are based on hypothetical avenues this software could take in the future rather than meeting set-in-stone goals and requirements because this iteration of the design is much more finished than the previous.

Part 3

Simulink (Assignment 1)

Module One: Input Parameters

Purpose:

The purpose of this module is to collect all parameters necessary for the pacemaker function and convert and pass (or else just pass) them through to the main function module of the code. It is imperative for this module that the variables are passed in the form that ius required for the rest of the program to execute effectively.

Black Box Behaviour:

The black box behaviour of all of these functions is very straight forward and has been described above. For each of them it expects the input of one variable that represents a user set desired value and the output will be this value converted in a way the program can use effectively.

Internal Behaviours:

For each of the conversions there is a specific process involved. These processes are described below.

Function:	Behaviour:
vent_amp to vent_pulse_amp	Apply gain of 100 and then convert to 32 bit unsigned integer
atr_amp to atr_pulse_amp	Apply gain of 100 and then convert to 32 bit unsigned integer
lowerRateIn to lowerInterval	Convert to double, divide by unit to normalize, apply gain, and finally convert to 32 bit unsigned integer
upperRateIn to upperInterval	Convert to double, divide by unit to normalize, apply gain, and finally convert to 32 bit unsigned integer

Module Two: Main Logic

Purpose:

The purpose of this module is to handle the main logic for deciding which states to operate in, such as AAI, AOO, VII, or VOO, and to execute the necessary logic for each mode to function properly.

Black Box Behaviour:

Our design incorporates black box behaviour by organizing the main module into five different states. Starting from an initial state, the system transitions to one of the four possible states (AAI, VVI, AOO, VOO) based on specific input parameters. The internal logic and functions within these states are hidden.

Internal Behaviours:

As for the global variables within this module, which act as state variables, include pace_atr, sense_atr, sense_on, ATR_CMP_DETECT, VENT_CMP_DETECT, ARP, VRP, atr_pulse_width, vent_pulse_width, lower_interval_rate. The first three are used as variables to determine the state the system will enter. For example, when pace_atr=0, sense_atr=0, and sense_on=1, the systems knows to enter the VVI state. These global variables are accessible across all five states and are used to guide the system into its specific mode. The rest of the global variables are used for managing state transitions within the logic of the chosen mode. For instance, in the VVI mode, the variables ATR_CMP_DETECT, ARP, lower_interval_rate, and atr_pulse_width are used in the timing of the pulses and deciding whether to charge or discharge the capacitor based on the detection of an atrial pulse. The same kind of logic appears across the other three modes, which also rely on the same global variables.

For our design, all the functions are private. Each mode (VII, AII, VOO, AOO) has specific logic that enable it to work as intended. This logic is determining if, and when to apply charging/discharging using the global variables. To charge or discharge, specific local variables are set to the values corresponding to the mode it is currently operating in. Apart from the global variables, none of this logic is being shared or reused as functions across the other modes.

Module Three: Output Pins

Purpose:

The purpose of this module is to receive the outputs from the main logic and assign them to the correct pins.

Black Box Behaviour:

The module incorporates black box behaviour by concealing the pin assignments. Internally, the outputs are connected to the appropriate pins and renamed, but on the outside the module simply receives the output from the main logic and outputs them.

Internal Behaviours:

The only logic happening within this module is setting the outputs received by the main logic to its corresponding pins. There are no functions being used, only the output variables from the main logic.

Simulink (Assignment 2)

Module One: Serial Communication/Egram

Purpose:

This module acts an interface for handling communication from the DCM. Upon receiving data, it processes the input and executes one of the three possible actions: updating parameter variables, run the egram function for monitoring, or sending the current parameters back.

Black Box Behaviour:

This module incorporates black-box behaviour by encapsulating all its internal logic within a single block. From an external perspective, the entire functionality is contained within a single block labeled “serial”, which houses all the logic described above. On the left side of this block, input data is received from the DCM via UART. The internal logic and decision-making based on this input data remains hidden behind the black-box design. On the right side is the output containing the updated parameters. Additionally, inside the “serial” is a state which calls the egram function. The logic for this function is defined outside of this module, which helps keep it hidden within the “serial” module. This is another example of incorporating black-box behaviour, which provides a clear and simplified design by hiding the internal logic.

Internal Behaviours:

The serial module utilizes two global variables, status and rxdata, which serve as key state variables. These variables control the transitions between the states and are accessible to all states within the module. The status variable is responsible for initiating the logic by exiting the standby state, while rxdata determines which of the three states to execute with the received data.

Regarding private functions, the module includes one: the egram function. This function is external to the module and remains hidden, with the module simply calling it to execute its operation.

Module Two: Rate Adaptive Logic

Purpose:

This module processes the data from the accelerometer and compares it to a threshold. Based on this comparison, it adjusts the desired pacing rate to a specific value. Once this rate is determined, it is sent to the main logic for use.

Black Box Behaviour:

We implemented black-box behavior in this module by organizing the logic into layers. Initially, we divided the functionality into two blocks: one that sets the rate-adaptive parameters to the correct data types and sends them to the second block, where the rate adaptation logic is executed. From an external viewpoint, there is just one block containing the parameters used by the rate adaptation logic, while the second block processes these parameters and outputs the corresponding pacing rate to the main logic. The internal workings of these blocks remain hidden.

Within the rate adaptation logic module, there are two additional blocks. The first block determines the activity level from the accelerometer data and adjusts the desired pacing rate accordingly. The second block manages the increase or decrease in the pacing rate to match the desired rate. Both of these blocks encapsulate their internal logic, and only the input/output flow of variables is visible when you're inside the "rate adaptive logic" module.

Internal Behaviours:

The global variables in this module are rate_adaptive, activity_level, activity_threshold, CPR, DPR, rate_down, and rate_up. These state variables are used to determine the transitions between states in the logic and are accessible by all states, which is why they are defined as global. The rate_adaptive variable checks whether the accelerometer is active, while activity_level and activity_threshold are compared to determine if the user is walking, running, or sprinting. The variables CPR and DPR are compared to decide whether to increase or decrease the pacing rate in order to reach the desired pacing rate (DPR).

DCM (Assignment 1):

Module 1:

a) This module is a framework for plotting electrogram data in real-time using matplotlib. It collects and stores data points (timestamps and voltages) and updates a graphical plot periodically. The module is designed to handle future customizations, such as different types of electrograms (atrium and ventricle).

b) The EgramData includes the following functions: __init__(self), add_data(self, voltage), get_data(self). The class EgramPlotter includes the following functions: __init__(self, title), animate(self, i), start_animation(self, interval). The derived Classes AtriumPlotter and VentriclePlotter have the function __init__(self). Global functions include plot_vent(), plot_atrium().

c) The black-box behavior of the functions in the module is as follows. In the EgramData class, the __init__ function takes no input and initializes an object with empty lists for timestamps and voltages while setting a counter to zero. The add_data function takes a single numeric voltage value as input and appends it to the voltage list while adding a corresponding timestamp to the timestamps list, automatically managing the data to retain only the last 20 entries. The get_data function takes no input and returns the current lists of timestamps and voltages.

   In the EgramPlotter class, the __init__ function initializes a plot window with a specified title and an instance of EgramData to manage plot data. The animate function, which receives a frame index as input, updates the plot by generating random voltage data and refreshing the display. The start_animation function takes an interval (in milliseconds) as input and starts the plot animation with updates occurring at the specified interval.

   The derived classes AtriumPlotter and VentriclePlotter inherit from EgramPlotter and initialize plots specific to atrium and ventricle electrograms, respectively. These classes use the parent functionality to set up the plot and manage data while providing an appropriate title.

   The global functions plot_vent and plot_atrium take no inputs. They create instances of VentriclePlotter and AtriumPlotter, respectively, and start their animations with a 100-millisecond update interval, providing dynamic visualizations for the respective electrograms. The module abstracts the internal mechanisms of data handling and plotting, focusing on providing easy-to-use interfaces for real-time electrogram visualization.

d) The global variables in this module are defined within the EgramData and EgramPlotter classes. In EgramData, self.counter tracks the number of data points

and increments with each call to add_data. The self.timestamps and self.voltages lists store timestamps and voltage values, respectively, keeping only the last 20 entries.

In EgramPlotter, self.fig and self.ax are matplotlib objects used to manage the plot, while self.title holds the plot's title. The self.data variable is an instance of EgramData, used to manage and update the timestamp and voltage data. These global variables facilitate real-time data plotting and ensure smooth visualization in the module.

e) This module doesn't involve any private functions

f) The internal behavior of each public function is as follows. In the EgramData class, the __init__ function initializes the object with an empty state, setting self.counter to 0 and creating empty lists for self.timestamps and self.voltages. The add_data function appends the current value of self.counter to self.timestamps and the input voltage to self.voltages. It increments self.counter and ensures that only the last 20 entries are maintained by truncating the lists when necessary. The get_data function returns the current state of self.timestamps and self.voltages without modifying them.

   In the EgramPlotter class, the __init__ function creates a new matplotlib figure and axes, stores the plot title, and initializes an EgramData object to manage the data. The animate function generates a random voltage, adds it to the data, and updates the plot by clearing the current axes and plotting the new data. The plot's title and labels are set, and the X-axis tick labels are rotated. The start_animation function starts the animation, updating the plot at regular intervals using FuncAnimation.

   In the derived classes AtriumPlotter and VentriclePlotter, the __init__ function calls the parent class's __init__ method, passing a specific title for either the atrium or ventricle electrogram. The global functions plot_vent and plot_atrium create instances of VentriclePlotter and AtriumPlotter, respectively, and start their animations with an interval of 100 milliseconds, managing the animations for the respective electrograms.

Module 2:

a) The purpose of this module is to handle the logic for managing user registrations and logins by reading from and writing to a CSV file (users.csv)

b) The module provides four public functions. The load_users() function loads user login information from a CSV file and stores it in the users_db dictionary. The save_user(name, password) function saves the current users in the users_db dictionary to the CSV file. The register_user() function registers a new user by validating input and checking if the username already exists. If the username is valid, the new user is stored in users_db and saved to the CSV file. Finally, the login_user() function validates the user's login credentials against the users_db dictionary and grants access if the credentials are correct.
c) The black-box behavior of the functions in this module is described as follows. The load_users() function reads the users.csv file and loads the usernames and passwords into the users_db dictionary. The save_user() function writes the data from the users_db dictionary into the users.csv file. The register_user() function checks if the username and password fields are filled, if the number of users does not exceed a limit, and if the username is unique. If the conditions are met, the new user is added to users_db and saved to the CSV file. The login_user() function compares the user's input against the users_db and grants access if the credentials match.
d) The global variables used in the module are users_db and FOLDER_PATH. users_db is a dictionary that stores login information, with usernames as keys and passwords as values. This dictionary is used across all functions in the module to manage user data. FOLDER_PATH is a string that stores the path to the folder where the users.csv file is located, and it is used to locate the file in multiple functions.
e) There are no explicitly private functions in this module.
f) The internal behavior of each public function is as follows: The load_users() function constructs the file path for users.csv, checks if the file exists, and reads the file using csv.reader(). It then adds the usernames and passwords to the users_db dictionary. The save_user() function writes the users_db dictionary into the users.csv file. The register_user() function counts the number of users, checks if the username is valid, and then adds the user to users_db, calling save_user() to update the CSV file. The login_user() function retrieves the user's input and validates it against users_db. If the credentials match, the user is logged in and redirected to the next part of the application.

Module 3:

a) The purpose of this module is to detect pacemaker devices connected through a specific serial port, retrieve their serial numbers, and store them if they are new.
b) The module provides three main public functions: get_serial(port), find_device(), and save_device(). The get_serial() function retrieves and returns the unique serial number from a given device port string. The find_device() function checks for a connected pacemaker device and returns its serial number if found. The

save_device() function checks whether the detected pacemaker device is new, and if so, saves its serial number to a CSV file.

c) The black-box behavior of each function is as follows: get_serial(port) takes a string containing port information, parses it to find a section starting with "SER=", and returns the serial number or None if no serial number is found. find_device() checks available serial ports for a device named "STM32 STLink" and retrieves its serial number. If no device is found, it returns None. save_device() checks for a connected device, retrieves its serial number using find_device(), and verifies if it's already saved in the CSV file. If the device is new, it appends the serial number to the file.
d) Global variables in this module include FOLDER_PATH, a string that stores the path to the folder containing the devices.csv file, and is used by find_device() and save_device() to locate the file.
e) The module does not have any explicitly private functions.
f) The internal behavior of each function involves handling device detection and serial number retrieval. For example, get_serial() splits the port string and extracts the serial number, while find_device() checks the serial ports for a device and retrieves its serial number. save_device() updates the devices.csv file by appending new devices and ensuring the file remains up-to-date.

Module 4:

a) This module manages pacemaker device connection detection and monitoring. It enables the system to recognize the first connected device, detect new or different devices, and alert the user when a different pacemaker device is connected. Additionally, it provides a user interface for selecting pacing modes and monitoring device connection status in real time.
b) The module contains four public functions: get_first_device(), alert_user(), update_device_label(), and mode_picker(). The get_first_device() function identifies the first connected pacemaker device and returns its serial ID. The alert_user() function alerts the user when a different pacemaker device is detected. The update_device_label() function monitors device connection status in real-time and updates the label with the current device information. The mode_picker() function opens a window where users can select pacing modes and view electrogram data.
c) The black-box behavior of each function is as follows: get_first_device() checks if a device is connected and retrieves its serial ID if available. alert_user() shows an alert message when a different device is detected. update_device_label() continuously checks the device connection status and updates the label with the current connection information, alerting the user if a different device is detected. mode_picker() creates a window with options for selecting pacing modes and monitoring device status in real time.
d) Global variables in this module include first_device_flag, a boolean that indicates if the first device has been identified; device_compare_flag, which ensures the user is alerted only once when a different device is connected; first_device, which stores

the serial ID of the first detected device; lbl_device, which is the label used to display the current device connection status; and root, the main GUI window that holds the mode selection interface and device updates.

e) This module does not contain any explicitly private functions.

f) The internal behavior of the functions is centered on device detection, user alerts, and maintaining real-time device connection status updates. For instance, get_first_device() calls save_device() to check for a device and sets the first_device_flag. The alert_user() function displays an information message and updates the device_compare_flag. update_device_label() checks device connection status, updating the label accordingly and alerting the user if necessary. Finally, mode_picker() initializes the GUI elements and starts the device monitoring loop with update_device_label().

Module 5:

a) This module initializes a CSV file for storing pacemaker device parameters and provides functionality to save specified pacing mode settings (VOO, AOO, VVI, or AAI) to the file. Each pacing mode requires specific parameters, which are formatted and written to the CSV accordingly.

b) The module includes two primary functions. The first, initialize_csv_file(), creates or overwrites a CSV file named pacing_parameters.csv, with headers to store pacing mode parameters. The second, save_settings(mode, lrl, url, amplitude, pulse_width, sensitivity=None, vrp=None, arp=None, pvarp=None, hysteresis=None, rate_smoothing=None), saves the pacing parameters for a selected pacing mode (VOO, AOO, VVI, or AAI). This function accepts a variety of parameters such as lower and upper rate limits, pulse amplitude, pulse width, and mode-specific parameters like sensitivity, ventricular refractory period (VRP), and atrial refractory period (ARP). Depending on the selected mode, only the relevant parameters are included in the saved row.c) Describe the Black-box Behaviour of Each Function (Part of the Module Interface Specification)

c) The black-box behavior of the functions is as follows. initialize_csv_file() takes no input but generates a new CSV file with appropriate headers: ‘Time’, ‘VOO’, ‘AOO’, ‘VVI’, ‘AAI’, and ‘Values’. This file serves as the storage for pacing parameters. On the other hand, save_settings() takes in the pacing mode and the corresponding parameters for that mode. It appends the relevant data to the CSV file in rows. If a parameter is not applicable to the selected mode, that column is left blank. For example, parameters like ventricular refractory period and sensitivity are only applicable to specific pacing modes (VVI and AAI), and will remain empty if not used.

d) This module does not contain any global variables. Both functions operate independently, using only the provided parameters and local variables for file paths.

e) This module has no explicitly private functions. All functions are accessible within the module and do not follow any convention indicating private status.

f) The initialize_csv_file() function defines the file path PARAMETER_FILE for pacing_parameters.csv, opens it in write mode (w), creating or overwriting the file if necessary. It then writes a header row with the columns: 'Time', 'VOO', 'AOO', 'VVI', 'AAI', and 'Values'. This function only writes the header and does not modify any internal state.

The save_settings() function opens pacing_parameters.csv in append mode (a), allowing new rows to be added. It checks the mode parameter to determine which values to save: for VOO and AOO, it saves the basic parameters (LRL, URL, amplitude, pulse width); for VVI and AAI, it includes additional parameters (sensitivity, VRP, ARP, PVARP, hysteresis, and rate smoothing) based on the mode. It appends the relevant data to the CSV file without altering internal state.

Module 6:

a) The purpose of this module is to provide a GUI-based settings interface for different cardiac pacing modes, specifically VOO, AOO, VVI, and AAI. Using the Tkinter library, it opens separate windows for each pacing setting where users can input various parameters, such as lower and upper rate limits, amplitude, pulse width, and other device-specific variables.
b) The module includes several public functions. open_voo_pacing_settings() opens a window for users to input parameters for the VOO pacing mode. Similarly, open_aoo_pacing_settings() opens a window for the AOO pacing mode, allowing users to input parameters like lower and upper rate limits, atrial amplitude, and pulse width. open_vvi_pacing_settings() opens a window for VVI pacing parameters, including additional fields such as sensitivity, VRP, and rate smoothing. open_aai_pacing_settings() works similarly for AAI pacing settings. The on_login_enter_key(event) function simulates a login button press when the "Enter" key is pressed.
c) The black-box behavior of each function is as follows: open_voo_pacing_settings() displays a window to accept user input for VOO pacing settings. Similarly, open_aoo_pacing_settings(), open_vvi_pacing_settings(), and open_aai_pacing_settings() each open respective windows to allow users to input relevant settings for each pacing mode. The handle_submit() function, embedded within each pacing function, validates and saves the entered data, providing a status message to indicate success or failure.
d) Global variables are not explicitly defined within the provided functions, though the Tkinter root object might be used as a global for the main window in other parts of the application.

e) The private function handle_submit() validates form inputs, checks for a connected device, and saves the settings. It operates similarly across each pacing settings window but differs based on the parameters required for each pacing mode.
f) The internal behavior of each function ensures proper window setup and input field creation. For example, open_voo_pacing_settings() creates a window with fields for lower rate limit, upper rate limit, ventricular amplitude, and pulse width. It then uses handle_submit() to validate inputs and, if successful, saves the settings using the save_settings() function. Similarly, each pacing settings window is designed with the appropriate parameters, validated, and saved. The on_login_enter_key(event) function simply binds the "Enter" key to simulate a click on the login button.

Module 7:

a) The purpose of this module is to create a graphical user interface (GUI) for a user management system within a "Pacemaker DCM" application. It allows users to either register a new account or log in to an existing one. The module uses a dictionary to store user data and relies on CSV file handling to persist this data, providing an easy-to-use interface with the help of the Tkinter library.
b) This module does not introduce any new public or private functions.
c) Since there are no newly defined functions in this module, there is no black-box behavior or internal function behavior to describe
d) It includes several global variables that are used across the module. users_db is a dictionary that stores user data, with usernames as keys and passwords (or other user information) as values. The first_device_flag and device_compare_flag are boolean flags used for managing the state of device connections, although they are not explicitly used within the provided code snippet.
e) This module does not include any private functions.
f) This module does not have any additional public functions are provided beyond those described above.

DCM (Assignment 2):

Module 1:

a) The module's purpose is to provide a framework for visualizing real-time electrogram (egram) data using Matplotlib and Tkinter. It integrates live data acquisition through serial communication and dynamically updates plots for atrium and ventricle electrograms. Enhancements include embedding Matplotlib plots into a Tkinter GUI, enabling user interaction, and safely managing resources such as open serial connections.

b) This module has the same public functions as assignment 1 except for the close_window() function.
c) The function within the EgramPlotter class were slightly altered from assignment 1. The black-box behavior of the __init__ function is to set up an interactive plotting interface for displaying electrogram data. It initializes a graphical user interface (GUI) window with an embedded Matplotlib plot and an "Exit" button. Inputs to the function include the title parameter, which determines the window title. The output is a fully initialized GUI for the plot, abstracting away the details of how the window, plot, and button are created.

   The start_animation(interval) has an input, interval (numeric) defined as the time in milliseconds between animation frames. The function dynamically updates the Matplotlib plot at the specified interval. The animate(i) function has an input of a frame index i (unused, but required by the animation framework). The function outputs a refreshed plot showing the latest timestamps and voltages.

   The close_window () function has no input and safely closes the Tkinter window and terminates the serial connection if active.
d) There are two new variables defined within the EgramPlotter class. The first is self.ser, which is initially set to None but is later expected to hold the serial connection object used to retrieve data from the pacemaker device. This variable is scoped within the EgramPlotter class and is used specifically in methods involving serial communication, such as animate and close_window. The second new variable is self.window, which is an instance of the tk.Tk() object, representing the Tkinter window that hosts the plot and other graphical user interface (GUI) elements. This variable is scoped within the EgramPlotter class and is used to manage the window for displaying the plot and handling user interactions, such as the "Exit" button.
e) This module does not include any private functions.
f) The __init__(self, title) function sets up the Matplotlib figure and axes, initializes a Tkinter window (self.window), and embeds the plot in the window. It assigns the provided title to the window and initializes self.data as an instance of EgramData to manage plot data. An "Exit" button is added, which calls close_window when pressed. This function prepares the GUI with the necessary components for displaying the electrogram plot.

   The start_animation(self, interval) function starts the plot animation, taking an interval parameter (in milliseconds) that defines the time between frames. It creates an animation using Matplotlib's FuncAnimation, which continuously updates the

plot at the specified interval. The animate(self, i) function updates the plot for each frame of the animation. It retrieves voltage data, adds it to self.data, and updates the plot by clearing the axes and re-plotting the new data. It also formats the plot with a title and axis labels.

The close_window(self) function safely closes the Tkinter window and the serial connection (if active). It checks if the serial connection (self.ser) is open and closes it before destroying the window to end the session.

Module 2:

a) The purpose of this code is to handle user management within an application. It provides functionality for user registration, login, and the storage of user data in CSV files.
b) There are two new functions within this module that were not deifnied in assignment two, the make_csv() function and the delete_csv() function.
c) The black-box behavior of the make_csvs() function is that it checks for the existence of a folder to store user-specific CSV files, creates the folder if it doesn't exist, and then generates individual CSV files for each user listed in a central users.csv file. It outputs a success or error message based on whether the folder exists and handles the creation of user-specific files. Similarly, the delete_csvs() function checks if the folder storing user-specific CSV files exists, and if so, removes it and all its contents, providing a confirmation message. If the folder doesn't exist, it outputs an error message.
d) There are no new global variables.
e) There are no private functions in this module.
f) The internal behavior of each function is as follows: make_csvs() constructs the path for the user_csvs directory, creates the directory if needed, reads the users.csv file, and creates individual CSV files for each user with a header row. The function ensures that only valid user rows are processed (those with two entries, representing a username and password). On the other hand, delete_csvs() constructs the path to the user_csvs folder, checks if it exists, and removes it entirely along with its contents using shutil.rmtree(). Both functions rely on file handling and directory creation mechanisms to perform their tasks efficiently.

Module 3:

This module is the same as the assignment 1 code.

Module 4:

a) This module handles the detection and monitoring of pacemaker device connections. It allows the system to identify the first connected device, track the connection of new or different devices, and notify the user when a different pacemaker device is detected. The module also offers a user interface for selecting pacing modes and continuously monitoring the device connection status in real time.
b) This module has three new functions, clear_window(), logout_user() and display_current_settings().
c) The clear_window() function takes no input and outputs a window with all its widgets removed, except for lbl_device, which is hidden but not destroyed. The logout_user() function also takes no input and outputs a cleared window, followed by the display of the login screen. The display_current_settings() function takes no input and either outputs the most recent settings saved for the current user or an error message. If the user's settings file is found and contains data, it retrieves and displays the most recent settings in a message box; otherwise, an error message is shown if the file is empty or unreadable.
d) Global variables in this module include root, which represents the main Tkinter window; lbl_device, the label displaying the current device connection status; and curr_user, which stores the username of the currently logged-in user.
e) There are no new private functions within this module.
f) Internally, clear_window() iterates over all widgets in the window, removing them, but ensures the lbl_device label is hidden without being destroyed. logout_user() first clears the window by calling clear_window(), then displays the login screen by calling show_login_screen(). display_current_settings() constructs the path to the current user's settings CSV file, reads the most recent settings if the file exists, and displays them in a message box. If there are issues with the file or no settings found, an error message is shown to the user. The mode_picker() function now includes buttons for VOOR, AOOR, VVIR, AAIR, a logout button that returns to the main screen, a message indicating which user is logged in and a current settings button that calls the display_current_settings() function.

Module 5:

a) This module handles the task of saving pacing parameters for different pacemaker modes (VOO, AOO, VVI, AAI, VOOR, AOOR, VVIR, and AAIR) to a CSV file. It stores these settings for each user, identified by their username, in a dedicated CSV file.
b) The save_settings function takes two parameters: mode, a string representing the pacing mode (such as 'VOO' or 'AOO'), and params, a dictionary that contains the settings related to that mode.

c) The save_settings() black-box behaviour involves the input of the mode and parameters, and the output is the updated CSV file with the new settings. It abstracts away the details of file handling, ensuring that only the necessary data for the selected mode is saved in the user's file.
d) The curr_user global variable is the main state variable in this module. It stores the username of the currently logged-in user, which is critical for determining where to save the pacing settings in the CSV file. This variable is accessed across the module to ensure that the pacing settings are stored under the correct user's file.
e) The module contains one private function, get_user_csv_path(username). This function is used to generate the file path for the current user's CSV file, combining the user's name (curr_user) with the appropriate folder path (user_csvs). This ensures that the pacing settings are written to the correct file for the currently logged-in user. The get_user_csv_path function is called within the save_settings function and is not intended to be accessed directly outside of it. It plays a crucial role in determining where to store the user's pacing settings but does not interact with or modify any state variables.
f) The save_settings function first determines the correct file path for the current user's CSV file using the get_user_csv_path private function. It then opens this file in append mode and writes the pacing parameters specific to the selected mode (e.g., 'VOO', 'AOO', 'VVI', etc.) into the file. The function checks which mode is selected and saves the corresponding parameters to the CSV file. If the mode requires additional parameters, such as sensitivity or refractory periods, those are included as well. The function ensures that only the relevant parameters for each mode are written, and the data is appended to the file without overwriting any existing settings. The get_user_csv_path function is used to generate the file path based on the current user's name (curr_user). It simply combines the folder path and the username to point to the appropriate CSV file for storing settings. The function does not modify any global or state variables directly but works based on the value of curr_user to ensure the settings are stored for the correct user.

Module 6:

a) The purpose of this module is to provide a GUI-based settings interface for different cardiac pacing modes, specifically VOO, AOO, VVI, and AAI. Using the Tkinter library, it opens separate windows for each pacing setting where users can input various parameters, such as lower and upper rate limits, amplitude, pulse width, and other device-specific variables.

b) This module has four new functions, open_voor_pacing_settings(), open_aoor_pacing_settings(), open_vvir_pacing_settings(), open_aair_pacing_settings().
c) The black-box behavior of each new function is as follows: open_voor_pacing_settings() displays a window to accept user input for VOOR pacing settings. Similarly, open_aoor_pacing_settings(), open_vvir_pacing_settings(), and open_aair_pacing_settings() each open respective windows to allow users to input relevant settings for each pacing mode.
d) There are no new global variables used in this module.
e) There are no new private functions within this module.
f) The internal behavior of each function ensures proper window setup and input field creation. For example, open_voor_pacing_settings() creates a window with fields for each parameter relevant to VOOR. It then uses handle_submit() to validate inputs and, if successful, saves the settings using the save_settings() function. Similarly, each pacing settings window is designed with the appropriate parameters, validated, and saved.

Module 7:

a) The purpose of this module is to provide a graphical user interface (GUI) for managing user accounts in the "Pacemaker DCM" application. It enables users to register new accounts or log into existing ones. The module stores user data in a dictionary and utilizes CSV files for data persistence, offering a user-friendly interface through the Tkinter library.
b) This module has one new function, show_login_screen()
c) The show_login_screen() function takes no direct input and its output is the creation and display of a login interface, including labels, input fields, and buttons, for the user to interact with for login or registration.
d) The global variables in this module include entry_name and entry_password, which are Tkinter Entry widgets for capturing the user's username and password. Additionally, btn_register and btn_login are Tkinter Button widgets for triggering user registration and login processes, while curr_user stores the current logged-in user's username. These variables are accessible across the module for handling user input and login management.
e) There are no private function within this module
f) The internal behavior of the show_login_screen function begins by calling clear_window() to remove any existing content from the window. It then sets up the window's properties such as title, size, and background color. The function creates and positions various UI elements including labels for the username and password fields, entry widgets for user input, and buttons for login and registration. These

elements are organized in a grid layout. The function also binds the "Enter" key to trigger the login process by invoking the on_login_enter_key function. Global variables for the username, password entry fields, and buttons are defined within this function for later use. The function ends by displaying the login screen, allowing the user to input credentials and interact with the interface.

Testing

Simulink (Assignment 1):

We went through many iterations of testing and design while building the Simulink model. We began with building an AOO using the same logic as in the final model. We used LEDs on the pacemaker to check what state we were in and when. We also tried a different approach to combining all the different modes by reusing the same sensing and pacing chart for each mode.

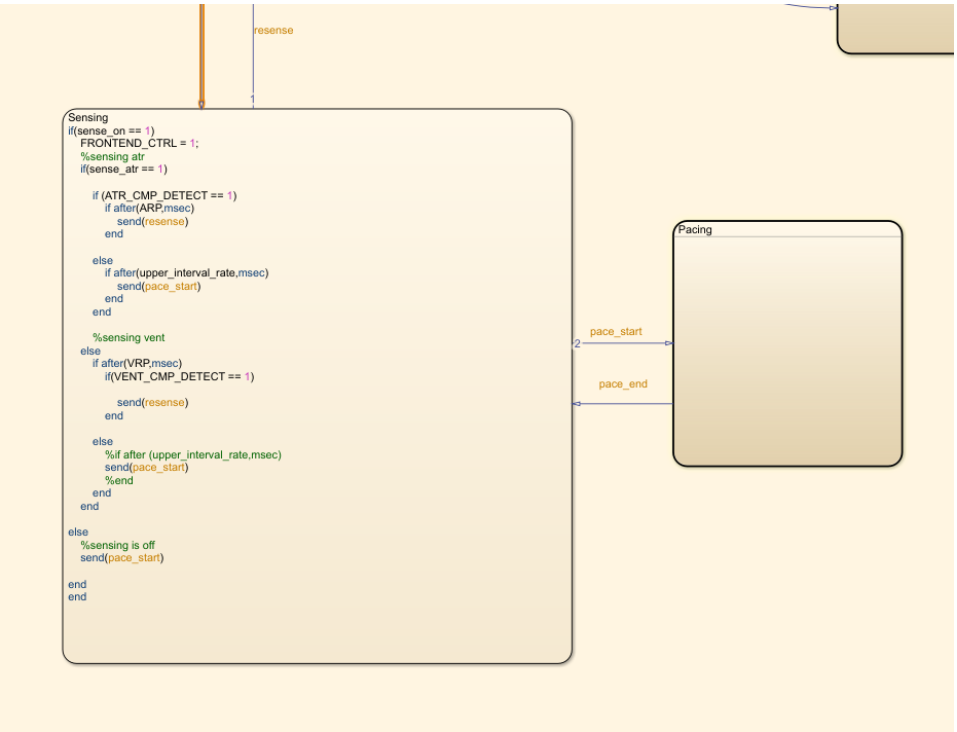


Figure 129: Old Sensing Function

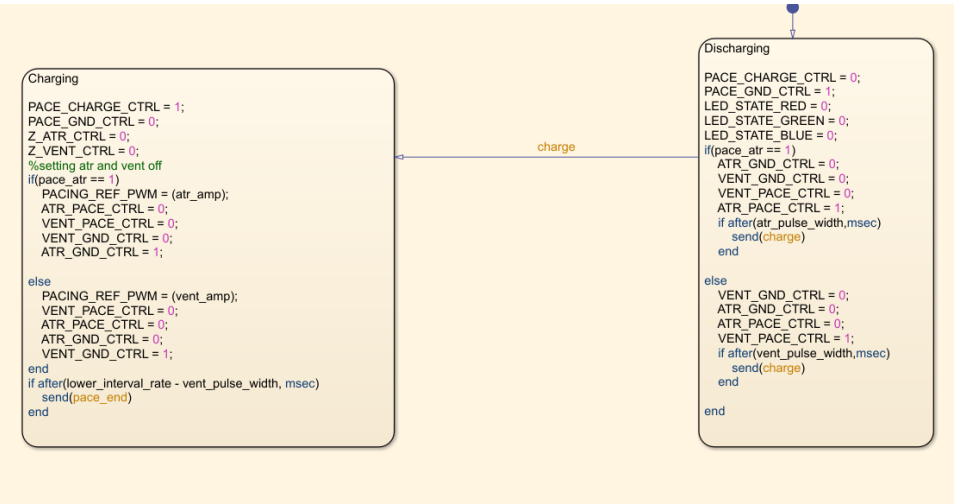


Figure 130: Old Pacing Function

This implementation worked for the most part but had some issues with the inhibiting modes where the timing would be off and sometimes a double pulse would happen.

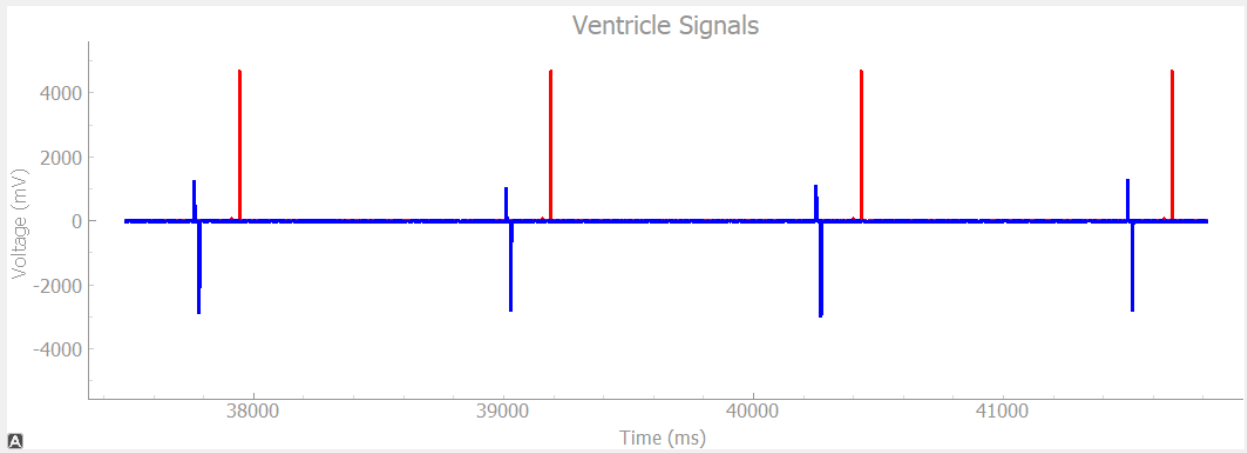


Figure 131: Double pace example

The entire model was moved to a more modular approach with each mode having their own sub-chart. The logic for the sensing had to change and used conditions to move between states instead of activities. This allowed for a different method to re-sense when a heartbeat was detected which fixed the issue.

Simulink (Assignment 2):

Whilst adding the new modes and implementing the new variables, we used test cases to verify the new modules were working as desired.

Rate adaptive Testing:

1. Purpose: The purpose of this test is to see if the accelerometer data is working with the different pacing state logic.
2. System Input: The input for this system will be shaking the device at different speeds
3. Expected Output: The expected output for not shaking the device will be a green LED turning on. For lightly shaking the device the blue LED will turn on. For heavily shaking the device the red LED will turn on.
4. Actual Output: When testing this the device began in the green LED state as expected. When shaken lightly the Blue led turned on. When the device was shaken heavily the red LED turned on.

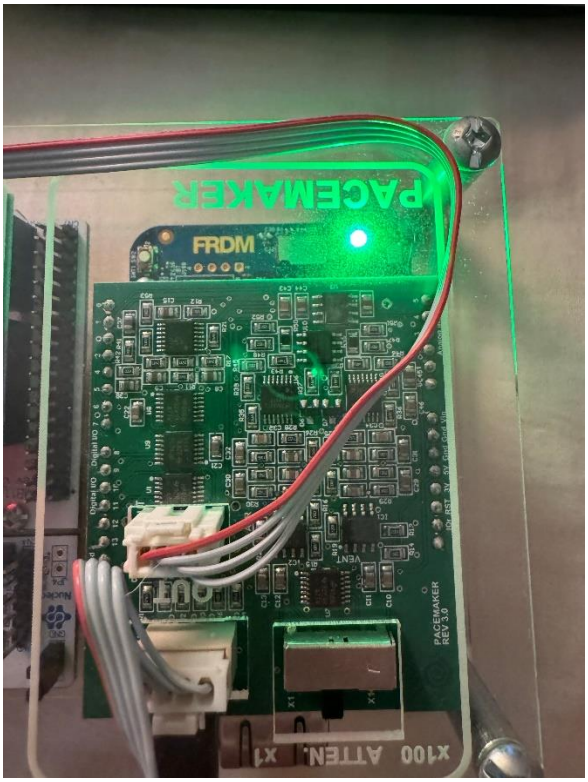


Figure 132: Resting State (Green)

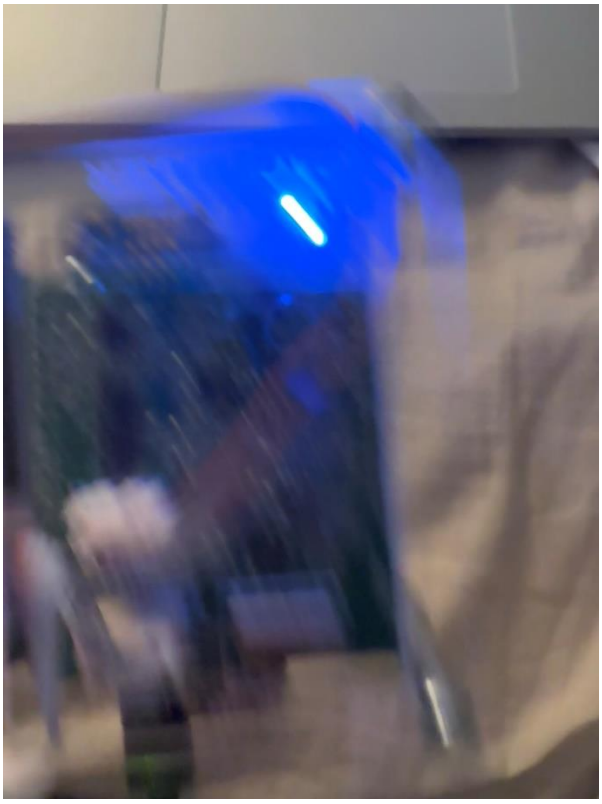


Figure 133: Running State (Blue)

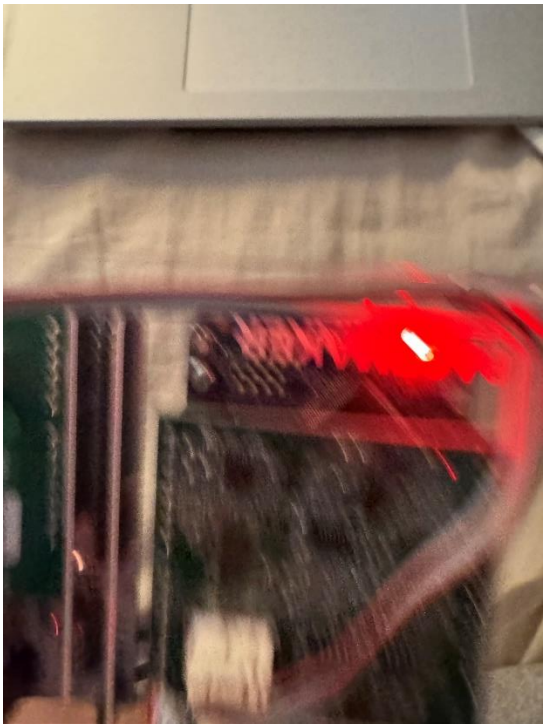


Figure 134: Sprinting State (Red)

5. Result: Pass, the actual output aligned with the expected output.

Testing serial communication DCM to Simulink:

1. Purpose: The purpose of this test is to see if the serial communication between python and Simulink are working.
2. System Input: The python file will send one uint8 byte to the device.
3. Expected Output: This will change the mode from AOO to VOO
4. Actual Output: The mode has changed from AOO to VOO

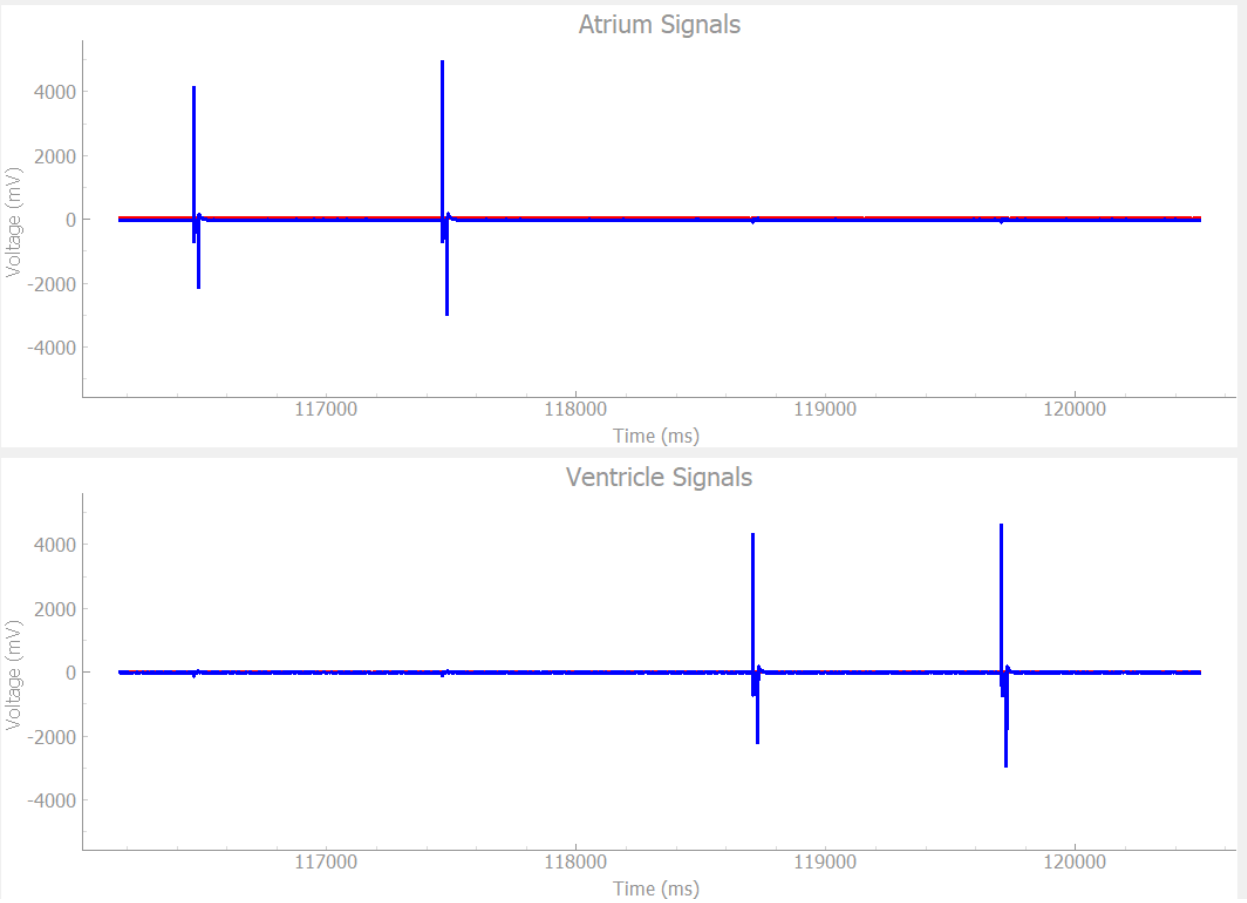


Figure 135: AOO to VOO

5. Result: Pass, the actual output aligned with the expected output.

Testing serial communication Simulink to DCM (byte packing):

1. Purpose: The purpose of this test is to see if the serial communication between Simulink and Python are working using the EGRAM.

2. System Input: The python file will ask to receive the data from the ventricle signal pin A1. The pin will be byte packed then serially transmit to python.
3. Expected Output: The EGRAM should plot the voltage from the ventricle signal pin.
4. Actual Output: The EGRAM plot is working

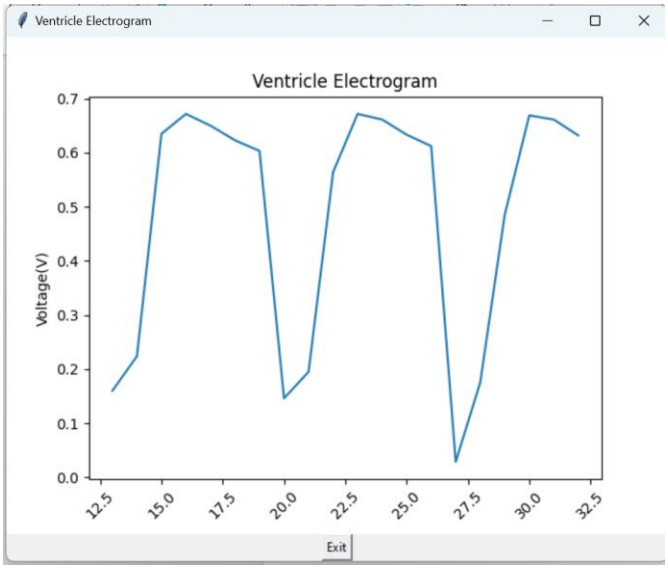


Figure 136: EGRAM test

5. Result: Pass, the actual output aligned with the expected output.

DCM (Assignment 1)

Throughout the development of our Device Communication Module (DCM), we implemented several testing processes to ensure functionality and user experience. Initially, we tested the device connection message by abstracting it to the terminal using print() statements. Once we validated that the message displayed correctly in the terminal, we integrated this functionality into the DCM, successfully displaying the message on the GUI. Additionally, we validated the egram plotting functionality by generating sample data sets and running separate plot functions, which confirmed that the code worked as expected before using random data. Each time we added new components to the DCM, we executed the code to verify that the layout remained functional and visually appealing, making necessary adjustments for spacing and alignment. Overall, all tests passed, indicating that the messaging, plotting, and GUI layout features of the DCM were robust and user-friendly. Moving forward, we recommend implementing automated tests for key functionalities, conducting user testing sessions for feedback, and performing performance evaluations under varying load conditions.

DCM (Assignment 2)

When going through the process of implementing the new requirements we devised test cases to ensure that the changes were working and were implemented correctly.

Serial Communication (Send):

The purpose of this test case was to familiarize ourselves with the method of serial communication between the DCM and pacemaker and to verify what format the data needed to be in to send successfully. The input for this test case began with one variable led_on which was set to 1 or 0. This was then converted to a byte and sent using the PySerial write() function. In Simulink a test model was created which used UART protocol to read the data written by the Python file to turn on or off an LED on the pacemaker board. The expected output when the variable led_on was set to 1 was for the red LED on the pacemaker board to turn on and when set to 0 the red LED would turn off. The result of this test case matched the expected, when the variable was set to 1 or 0 the red LED was turned on and off respectively. The output of this test case was a pass.

```python
import serial
import serial.tools.list_ports
import struct
sync = b'\x16' # Needed to send data

for port in ports:
    print(port.description)
    # Check if pacemaker is connected
    if "JLink" in port.description:
        currPort = port.device
        break

if currPort:
    print(f"Device is connected to: {currPort}")
else:
    print("Device not found.")

ser = serial.Serial(currPort, 115200, timeout = 1)

led_on = 1
red_led = struct.pack('B', led_on)

data_to_send1 = sync + b'\x55' + red_led
bytes_written = ser.write(data_to_send1)
print(f"Sent {bytes_written} bytes: {data_to_send1}")
```

Figure 137 Serial communication writing test case Python code

Serial Communication (Read):

This test case was used to find out and confirm what format data would be in when read from the pacemaker. Since the data we would be reading was egram data, the input for this test case was one variable send_data, which was assigned to be a float value. First this value was written to the pacemaker, and when received on Simulink it was assigned to a

variable. After writing, we would then read what the variable was by using the read() function and printing out what the data read was. The send_data variable was initialized to be 6.2. The expected output for the data read was 6.2. When reading and printing the data, the variable was printed in byte form, but when unpacked it successfully printed out the expected value. The result of this test was a pass since we confirmed our method of reading data from the pacemaker and got the output we expected.

```
import serial
import serial.tools.list_ports
import struct
import time
sync = b'\x16' # Needed to send data

for port in ports:
    print(port.description)
    # Check if pacemaker is connected
    if "JLink" in port.description:
        currPort = port.device
        break

if currPort:
    print(f"Device is connected to: {currPort}")
else:
    print("Device not found.")

ser = serial.Serial(currPort, 115200, timeout = 1)

send_data = 6.2
send_data = struct.pack('f', send_data)

data_to_send1 = sync + b'\x55' + send_data
bytes_written = ser.write(data_to_send1)
print(f"Sent {bytes_written} bytes: {data_to_send1}")

time.sleep(2)

data_to_send2 = sync + b'\x22' + send_data
ser.write(data_to_send2)
read_data = ser.read(4) # read 4 bytes (float = 4 bytes)
print("data read: ", read_data)
read_data = struct.unpack('f',read_data)
print("unpacked data: ",read_data)
ser.close()
```

Figure 138 Serial communication reading testing Python code

```
Device is connected to: COM4
Sent 6 bytes: b'\x16Uff\xc6@'
Sent 6 bytes: b'\x16Uff\xc6@'
data read:  b'ff\xc6@'
data read:  b'ff\xc6@'
unpacked data:  (6.199999809265137,)
```

Figure 139 Output of test code

Assurance Case

This assurance cases shown below reviews the path and the taken to identify the possible hazards and ensure that there is a method to handle them.

Simulink

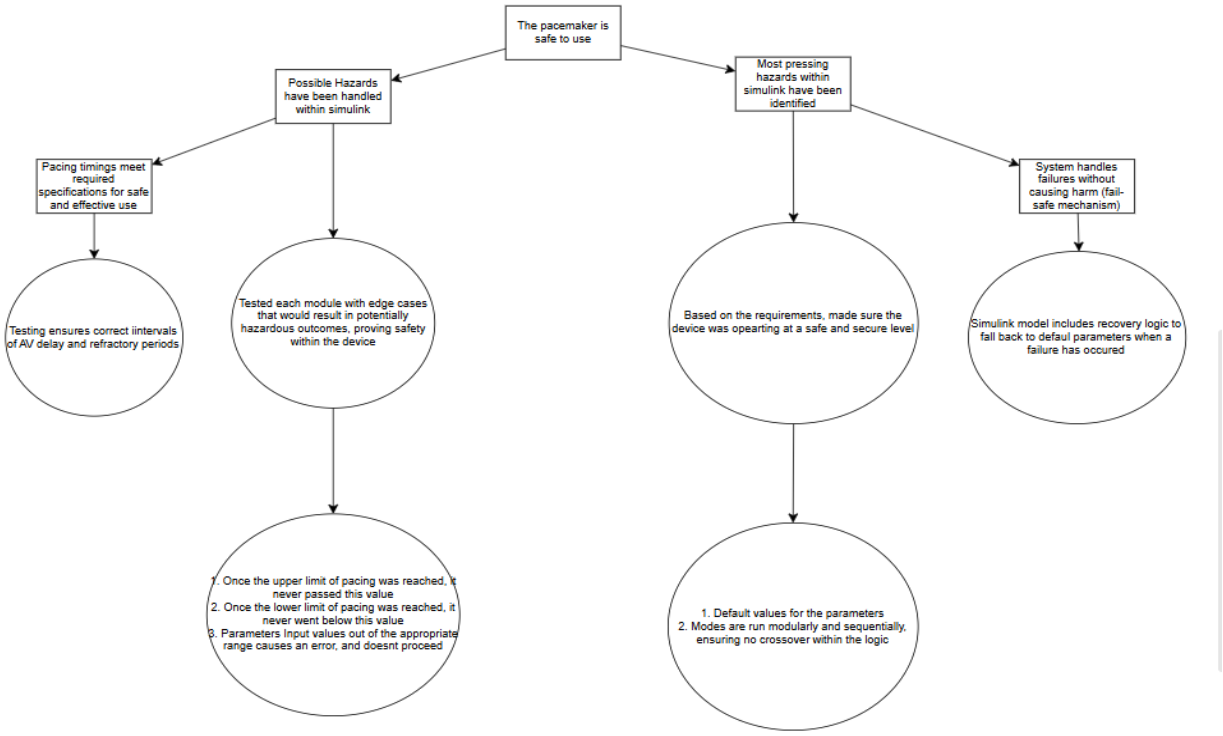


Figure 140: Assurance Case for Simulink Model

DCM

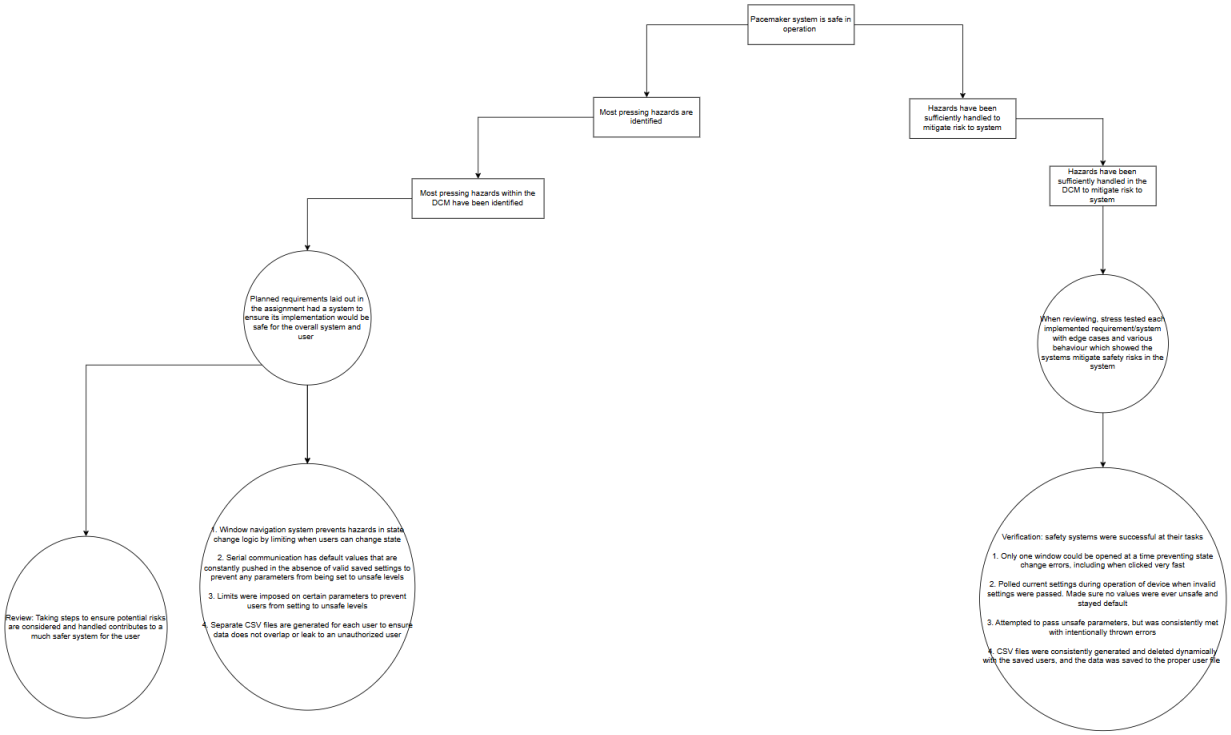


Figure 141 Assurance case