SFWR ENG 3K04 ASSIGNMENT 1

SFWR ENG 3K04 – Software Development

Professor: Thomas Chiang

Date of Submission: October 25, 2024

Team: Wednesday_The Redliners

John Sarga_ Sargaj

Jiayi Yang (Max)_yangj285

Zihao Zhou (Bill)_ zhouz247

Changhai Bian_ Bianc1

Mustafa Hassan_ hassam75

PACEMAKER – Simulink

1. Requirements

The pacemaker Simulink model is required to simulate the following four modes with specific programmable user parameters, ensuring each mode operates as intended for accurate pacing.

AOO Mode (Atrial Asynchronous Pacing):

- **Description**: This mode provides consistent pacing to the atrium at a fixed rate, independent of natural atrial activity. There is no sensing involved, so the pacemaker does not inhibit pacing based on intrinsic activity.
- **Key Parameters**:
  - **Atrial Pulse Width**: Duration of each pacing pulse delivered to the atrium.
  - **Atrial Amplitude**: Voltage amplitude of the atrial pacing pulse.
  - **Pacing Rate**: Frequency of atrial pulses, configurable based on the user-defined Lower Rate Limit (LRL).

VOO Mode (Ventricular Asynchronous Pacing):

- **Description**: The pacemaker paces the ventricle at a fixed rate without sensing natural ventricular beats. Like AOO, it delivers pulses independently of intrinsic activity.
- **Key Parameters**:
  - **Ventricular Pulse Width**: Duration of each pacing pulse delivered to the ventricle.
  - **Ventricular Amplitude**: Voltage amplitude of the ventricular pacing pulse.
  - **Pacing Rate**: Frequency of ventricular pulses based on the user-defined LRL.

AAI Mode (Atrial Demand Pacing):

- **Description**: In this mode, the pacemaker paces the atrium only when it detects an absence of natural atrial activity within a specific interval. It inhibits pacing if intrinsic atrial beats occur within the pacing interval.
- **Key Parameters**:
  - **Atrial Pulse Width**: Duration of each atrial pacing pulse.
  - **Atrial Amplitude**: Voltage amplitude of the atrial pacing pulse.
  - **Lower Rate Limit (LRL)**: Minimum rate to pace the atrium if no natural beats are detected.
  - **Atrial Refractory Period (ARP)**: Interval during which the pacemaker will ignore sensed events in the atrium to prevent double-counting.

VVI Mode (Ventricular Demand Pacing):

1. **Description**: The pacemaker paces the ventricle only if no natural ventricular activity is detected within a set pacing interval. It will inhibit pacing when it detects intrinsic ventricular activity.
2. **Key Parameters**:
    - **Ventricular Pulse Width**: Duration of each ventricular pacing pulse.
    - **Ventricular Amplitude**: Voltage amplitude of the ventricular pacing pulse.
    - **Lower Rate Limit (LRL)**: Minimum rate to pace the ventricle if no natural beats are detected.
    - **Ventricular Refractory Period (VRP)**: Interval during which the pacemaker will ignore sensed events in the ventricle to avoid multiple detections.

2. Design Decisions

2.1 Architecture

The overall architecture of the pacemaker's software, as depicted in the Simulink design shown in Figure 1, is composed of three main modules: **Input Module**, **MainStateFlow Module**, and **Output Module**. These design decisions were made to ensure modularity, flexibility, and ease of maintenance while providing clear separation of functionality and responsibilities.

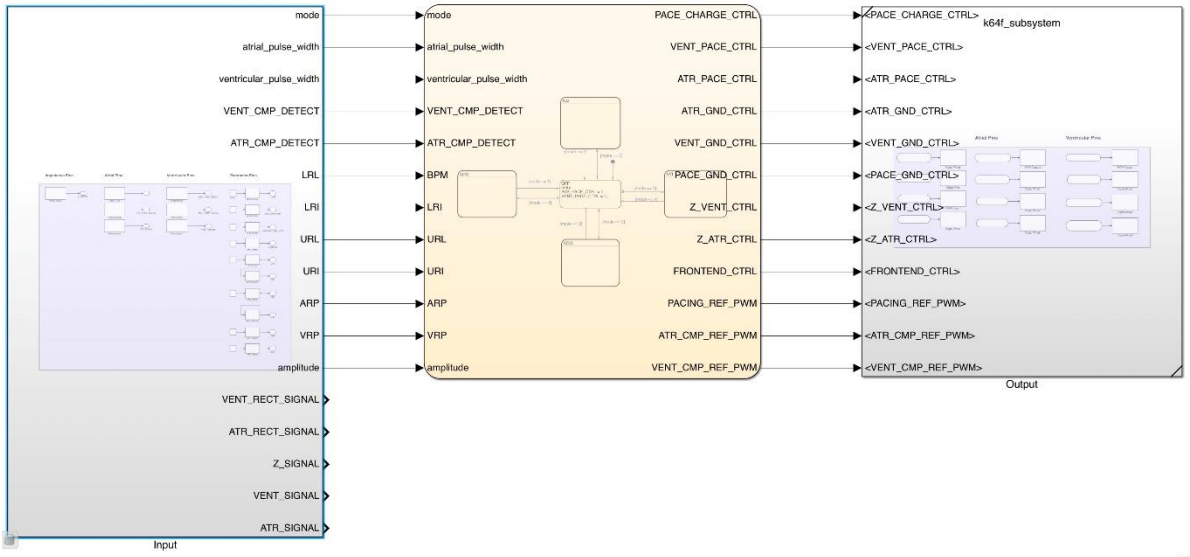


Figure 1. Simulink Block Diagram

Input Module

The Input Module is responsible for gathering all incoming signals, both from the heart (atrial and ventricular signals) and user-configurable parameters. This module as shown in Figure 2 ensures that data is correctly processed before it reaches the decision-making module.

This modular design clearly separates real-time monitoring from user-defined settings. This decision allows for easier testing and validation of each input source, ensuring that each signal is handled accurately. Additionally, maintaining modularity in input handling helps with scalability, allowing future modifications (e.g., adding new input parameters) without affecting the entire system.

- **User-Configurable Parameters with Limits**:
    - **Mode:** Ranges between **0-4.**
    - **Lower Rate Limit (LRL)**: Ranges between **30-175 BPM**.
    - **Upper Rate Limit (URL)**: Ranges between **50-175 BPM**.
    - **Pulse Width**: Ranges between **0.1-1.9 ms**.
    - **Amplitude**: Configurable between **1-100**.
    - **Atrial Refractory Period (ARP)**: Set between **150-500 ms**.
    - **Ventricular Refractory Period (VRP)**: Set between **150-500 ms**.

The Input Module ensures these parameters remain within the specified limits, guaranteeing safe operation and proper pacing behavior. It clearly separates real-time heart signals from user-configurable inputs for accurate control and functionality.

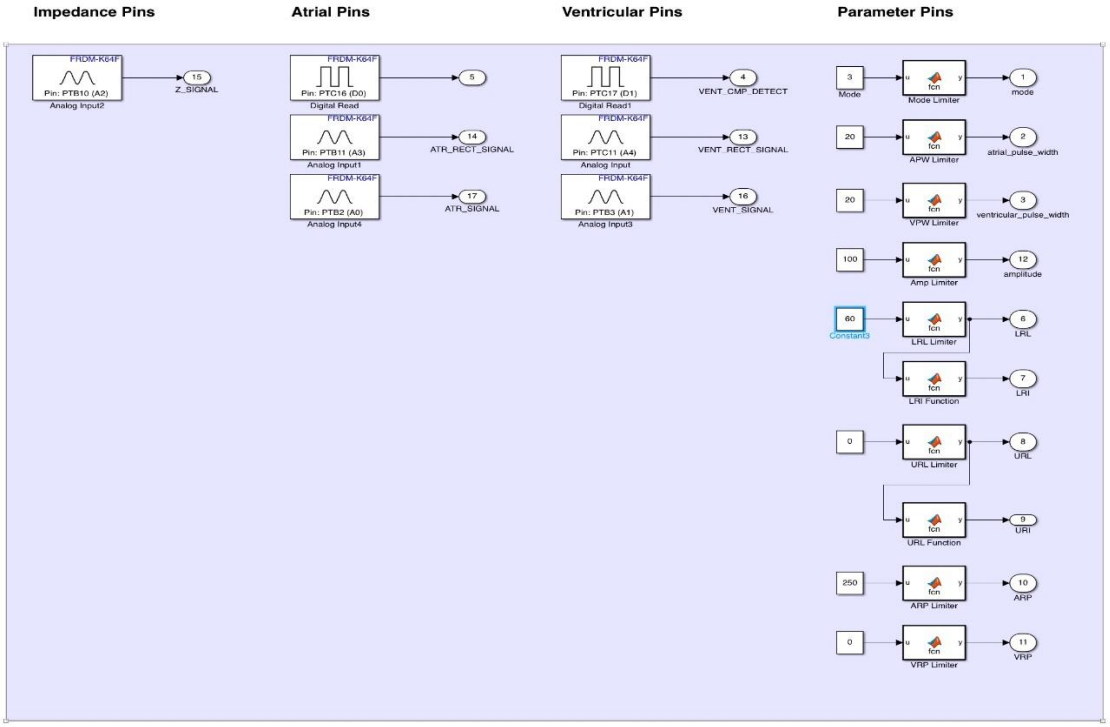


Figure 2. Input Diagram

MainStateFlow Module:

The MainStateFlow Module serves as the core logic processor and is responsible for decision-making based on the inputs from the Input Module as shown in Figure 3. It determines which pacing mode to activate (AOO, VOO, AAI, or VVI) and controls pacing actions, such as triggering pacing pulses.

This separation enhances modularity and ensures that the decision-making process is not

entangled with hardware operations, making it easier to test and verify the pacing logic independently. It also simplifies updates—if the decision logic changes (e.g., a new mode is introduced), it can be modified without affecting the hardware control aspects of the system.

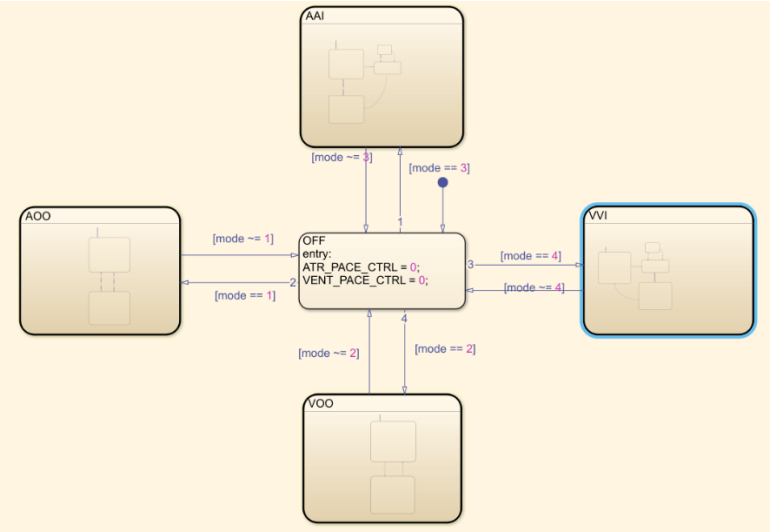


Figure 3. Stateflow Chart for Pacemaker Modes

Output Module:

The Output Module converts the logical decisions from the MainStateFlow into physical signals that control the pacemaker's hardware. It translates the pacing actions into control signals, such as activating the appropriate electrodes for pacing.

The Output Module is dedicated solely to hardware control, translating logical actions (e.g., PACE_CTRL signals) into hardware commands.

By isolating hardware control from logic processing, the Output Module provides a clear interface between the software logic and the hardware. This modular design makes it easier to maintain and update the hardware interface without affecting the core logic. For example, if a different hardware platform were used, only the Output Module would need to be adapted, leaving the MainStateFlow intact.

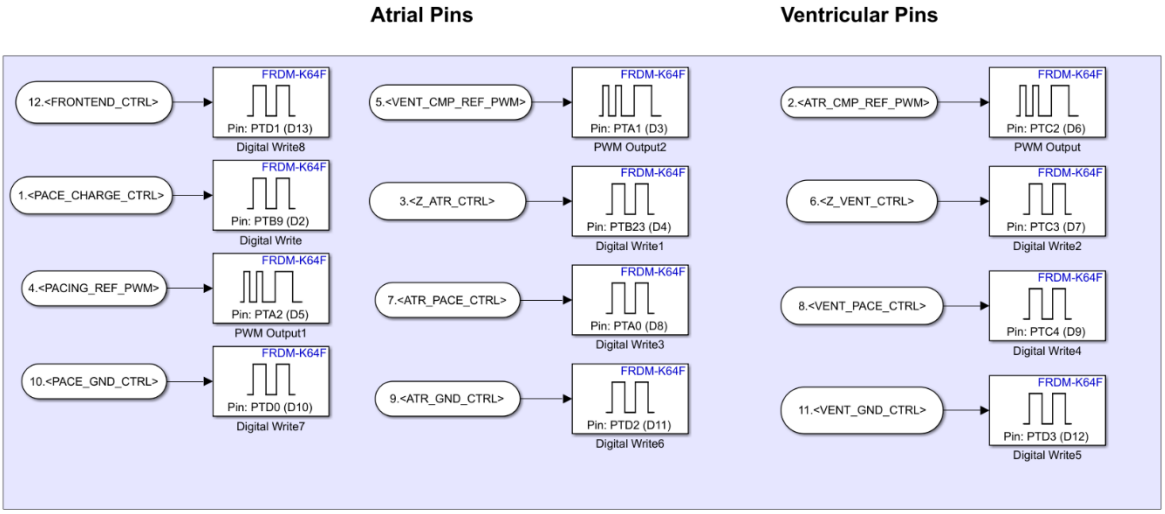


Figure 4. Detailed Pin Assignment Diagram

2.2 Mode Selection

AOO and VOO (Asynchronous Modes): These modes provide fixed-rate pacing to the atrium (AOO) or ventricle (VOO) without sensing natural heart activity. They are simple to implement and require minimal logic, making them reliable in cases where continuous pacing is required, regardless of heart activity. The simplicity of asynchronous pacing ensures reliability in patients who require continuous support, regardless of intrinsic heartbeats, and minimizes the need for complex sensing and timing logic.

AAI and VVI (Demand Modes): These modes monitor natural heart activity and only deliver pacing pulses when no intrinsic beat is detected. AAI focuses on the atrium, while VVI focuses on the ventricle. Demand modes conserve battery life by pacing only when necessary. This approach also allows the pacemaker to work harmoniously with the patient's natural heart activity, avoiding unnecessary pacing when the heart is functioning adequately on its own.

Flexibility: Implementing these modes separately within the MainStateFlow allows for easy switching between modes, depending on the patient’s needs. Each mode can be adjusted or updated without interfering with other modes, making the system adaptable.

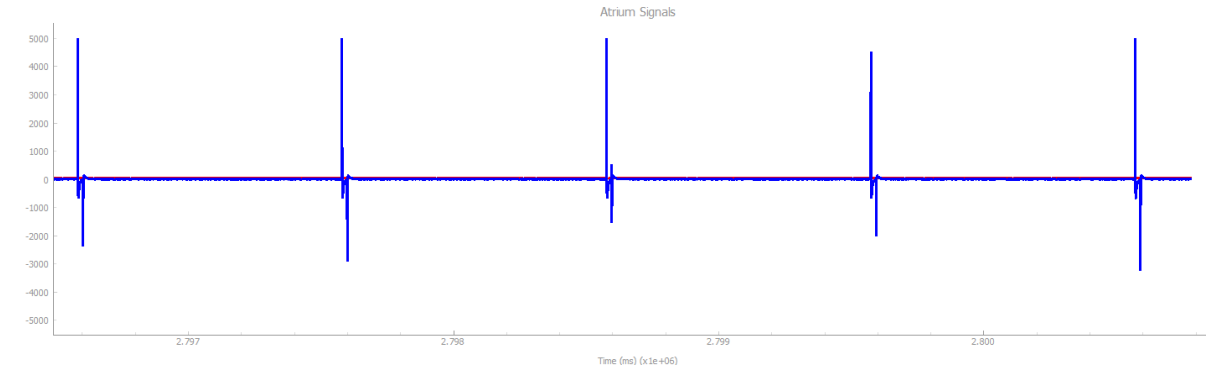
Safety and Efficiency: The modes were implemented with clear separation between asynchronous and demand-based modes, ensuring that each type of pacing is optimized for either continuous or on-demand support. This helps maintain patient safety while ensuring efficient use of the pacemaker's battery and resources.

3. Validation and Verification

Validation was completed on each of the 4 possible pacing modes to make sure that they behaved as expected.

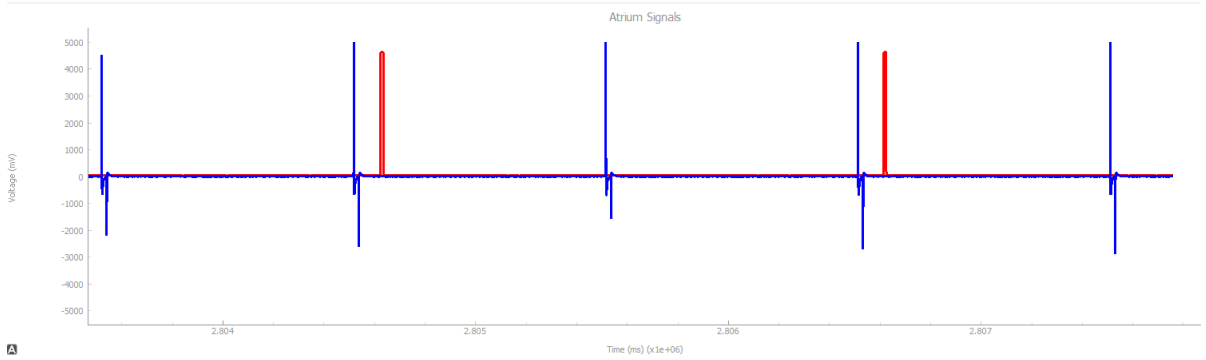
AOO

There were 2 sets of tests done on this mode to make sure that it is working. The first is to turn off the natural heart rate and set the pacemaker’s target to 60 BPM. This test was passed successfully as the pacemaker paced every second.

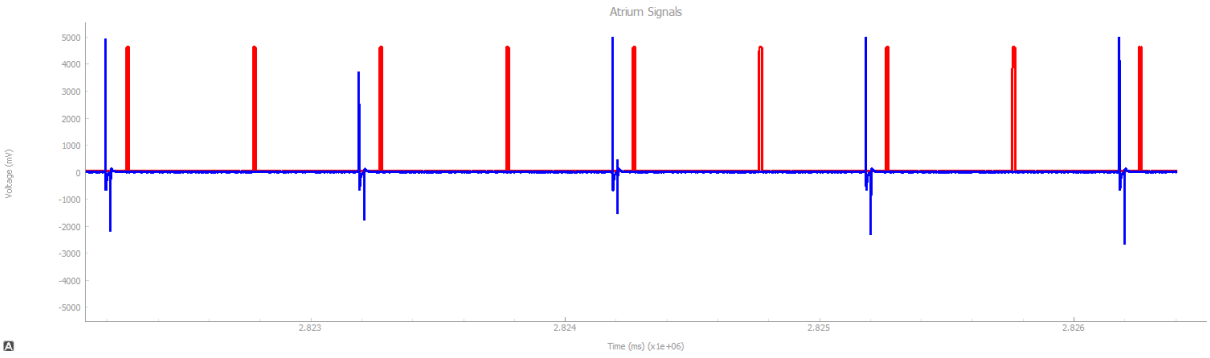


5 Graph of the atrium showing artificial pacing at 60 BPM

Then the next set of tests consisted of turning on the natural heart rate with a BPM that was either much higher or lower than the pacemaker's target of 60 BPM. This is to make sure that the pacemaker doesn't change its actions based on the natural heart rate in this mode. Both tests passed as shown below:



6 Graph of atrium with natural heart rate set to 30 BPM

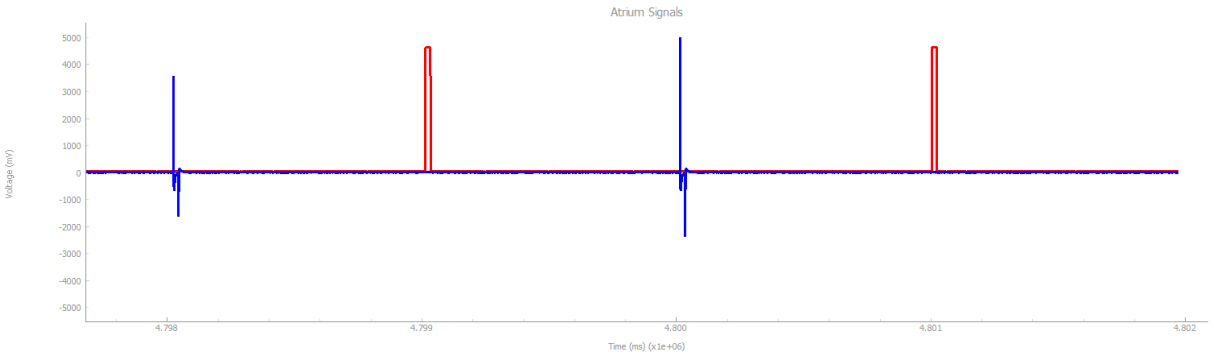


7 Graph of atrium with natural heart rate set to 120 BPM

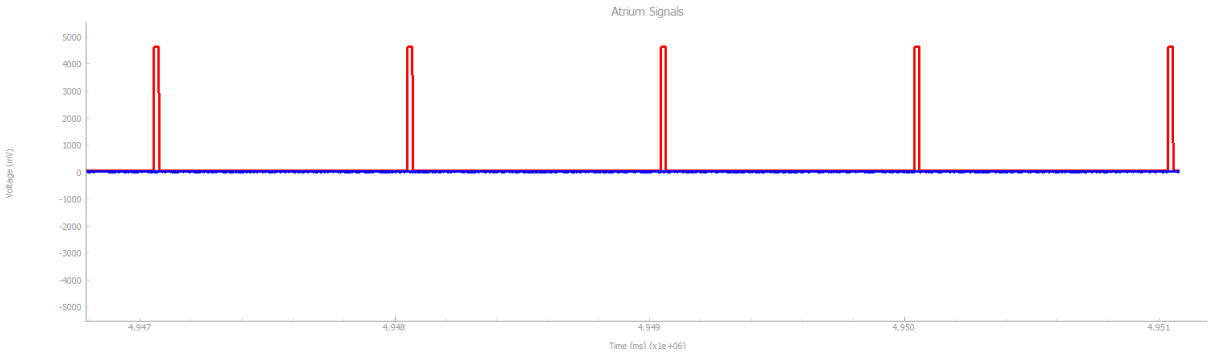
AAI

To test the sensing capabilities of this mode, the target heart rate was set to 60 BPM and natural heart rate was first set to 30 BPM, which is much lower than the target. It is expected that

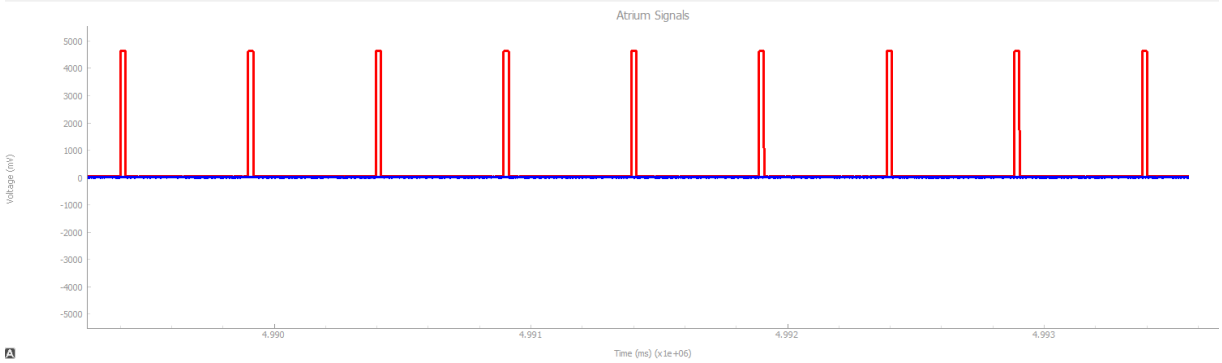
pacing should occur. It was then set to 60 BPM, which is the point where pacing should stop since the target has been reached. For the last test, the natural heart rate was set to 120 BPM, to make sure that that no pacing occurs when the natural heart rate is significantly higher than the target. All three of these tests have passed as shown in the graphs below.



8 Graph of Atrium with natural heart rate set to 30 BPM



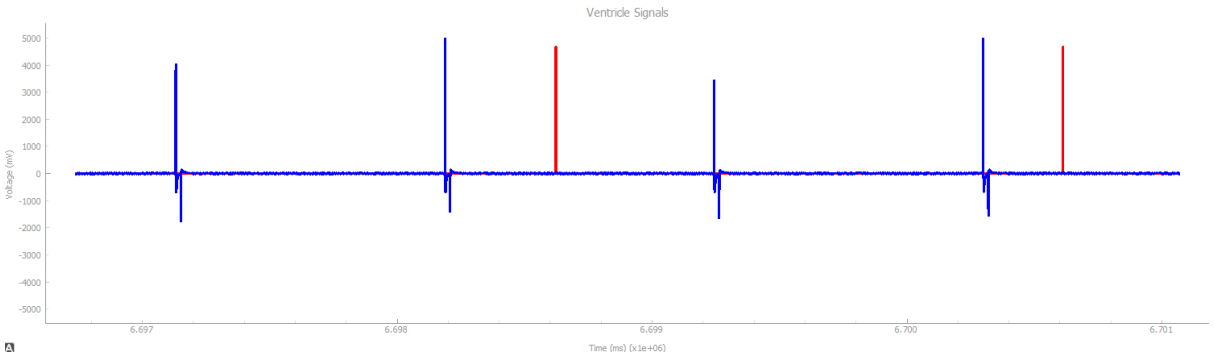
9 Graph of Atrium with natural heart rate set to 60 BPM



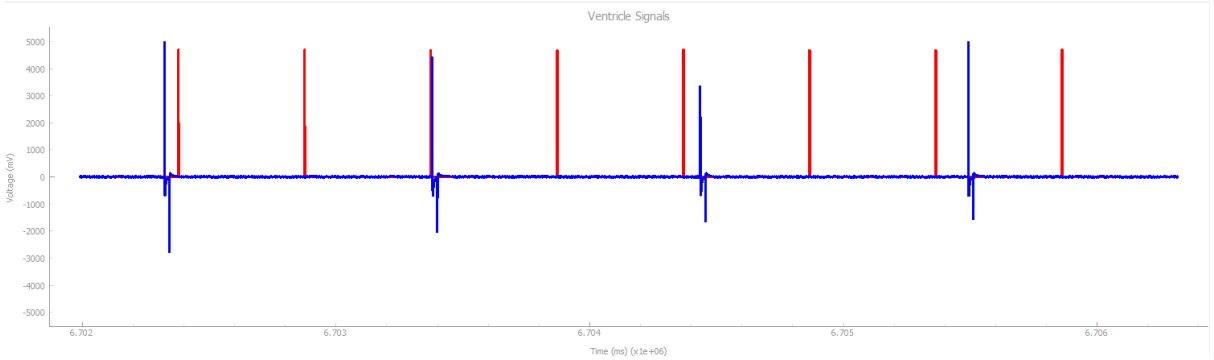
10 Graph of Atrium with natural heart rate set to 120 BPM

VOO

The testing for this mode is the same as the testing for AOO, since the expected pacing for the ventrecle is the same. This mode passed both tests and was not affected by the natural heart beat, as shown in the graphs below.



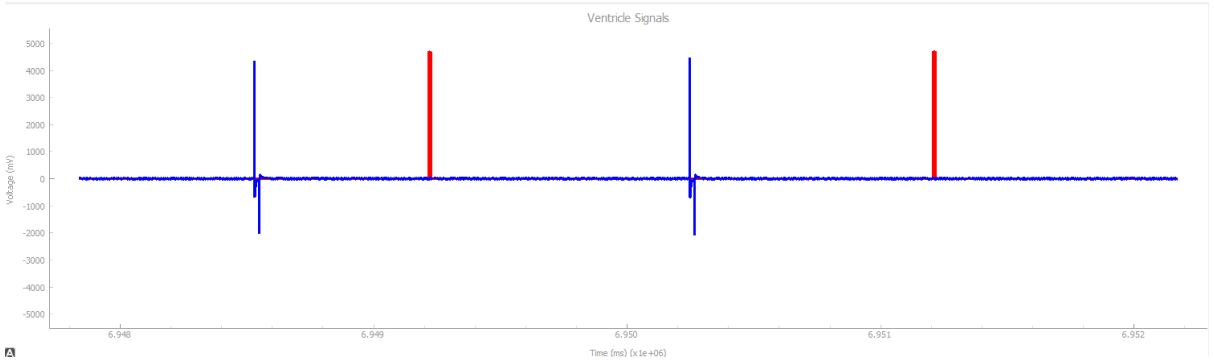
11 Graph of ventricle with natural heart rate set to 30 BPM



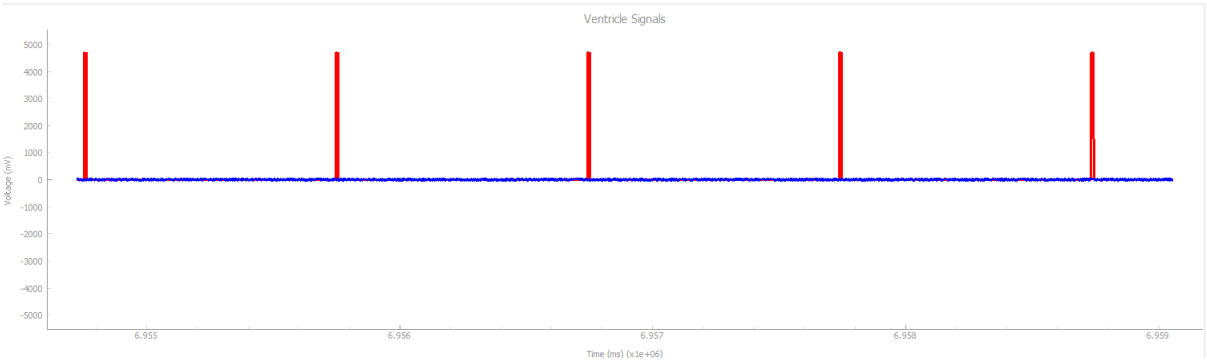
12 Graph of ventricle with natural heart rate set to 60 BPM

VVI

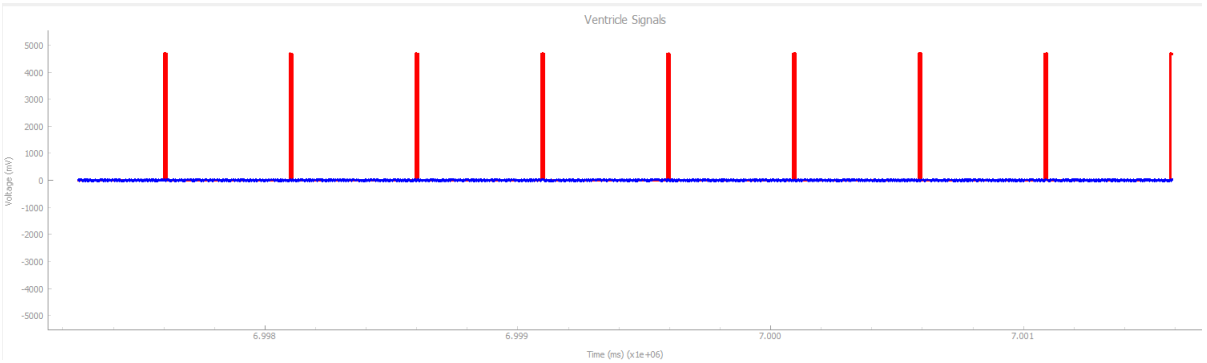
The testing for this mode is the same as the testing for AAI, since the expected pacing for the ventrecle is the same. This mode passed all the tests and was able to sufficently react to changes in the natural heart rate as shown in the graphs below:



13 Graph of Ventricle with natural heart rate set to 30 BPM



14 Graph of ventricle with natural heart rate set to 60 BPM



15 Graph of ventricle with natural heart rate set to 120 BPM

Input Limiters

The input limiter blocks connected to the inputs were tested to make sure that they limit the input to safe values, this was done by attaching a display to the output of the limiters and moniting the output as values that were above and below the safety threshold were set as inputs. All of the limiters behaved as expected and did not allow the inputs to reach unsafe values.

4. Requirements Changes

As the pacemaker system evolves, several changes to the current requirements may be necessary to accommodate future advancements and specific patient needs. One likely change is the addition of more advanced pacing modes, such as rate-adaptive pacing. This would adjust the pacing rate dynamically based on physical activity, requiring new sensors and the introduction of adjustable parameters like the Upper Rate Limit (URL).

Another potential change could involve increasing the flexibility of existing parameters, such as allowing real-time adjustments to pulse width, amplitude, or refractory periods based on feedback from the heart's performance. This would enable the pacemaker to respond dynamically to the patient’s condition without requiring manual reconfiguration.

Additionally, the current safety limits for programmable parameters like the Lower Rate Limit (LRL), Ventricular Refractory Period (VRP), and Atrial Refractory Period (ARP) may need to be extended or refined. This would ensure that the pacemaker can operate effectively for a broader range of patients with unique cardiac conditions.

Finally, the system might need to incorporate features like data logging or telemetry, enabling real-time data transmission for post-implantation monitoring and analysis. This would introduce new requirements for handling and transmitting patient data efficiently and securely.

5. Design Decisions Changes

The **MainStateFlow Module** currently processes inputs and decides the appropriate pacing actions for four distinct modes: AOO, VOO, AAI, and VVI. With future advancements, new pacing modes, such as rate-adaptive pacing, could be introduced to respond to physical activity levels or other physiological indicators. Implementing such modes would require adding or modifying state transitions, pacing logic, and potentially integrating new inputs (such as activity sensors) within the MainStateFlow. Additionally, as battery efficiency and safety become increasingly important, the logic within this module may need adjustments to optimize power consumption and pacing accuracy.

6. Module Clarification

AOO Mode State Machine

State 1: AOO_Charge

The pacemaker prepares to deliver a pacing pulse to the atrium. It charges the pacing circuit and ensures the ground is properly connected, while setting up the necessary internal configurations for atrial pacing. During this stage, the pacemaker is not actively pacing, but it's waiting for the interval to pass, which is calculated based on the pacing rate (Lower Rate Limit, LRL) minus the duration of the upcoming pulse. This ensures that the pacing pulse is delivered at the correct time.

Transition: Once the timing is right (based on the configured pacing rate), the system moves to the next state, where it delivers the pulse.

State 2: Atrial_Pacing

In this state, the pacemaker actively delivers a pacing pulse to the atrium. It briefly applies a controlled electrical signal, which stimulates the atrial muscle to contract. Once the pulse is delivered, the system turns off the pacing pulse and disconnects the ground connection to the atrium, completing the pacing action.

Transition: After the pulse is delivered, the system transitions back to the charging state to prepare for the next pacing cycle, repeating the process based on the programmed rate.

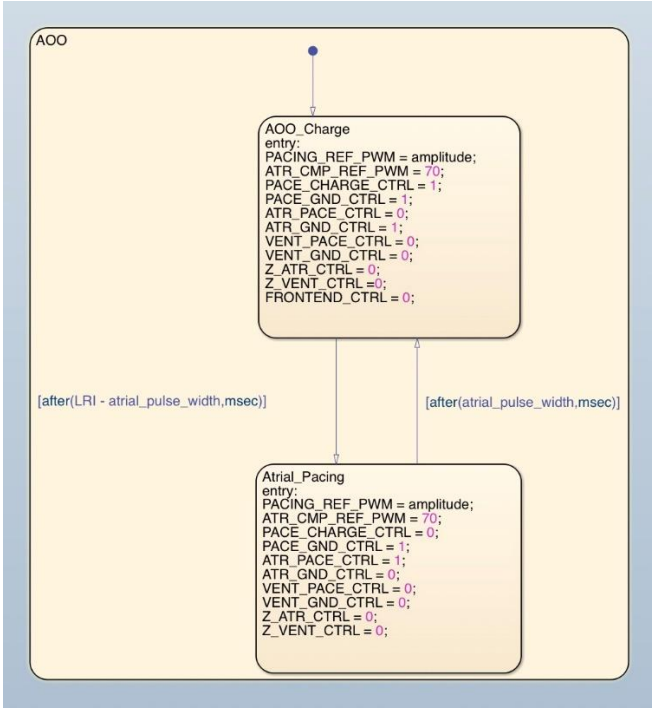


Figure 16. AOO Mode State Machine

AAI Mode State Machine

State 1: AOO_Charge

The pacemaker charges the atrial pacing circuit and waits for the appropriate interval based on the pacing rate. During this time, it prepares to deliver a pulse if no intrinsic atrial activity is detected.

Transition: The state transitions to **inhibited_ARP**.

State 2: inhibited_ARP

This state represents the Atrial Refractory Period (ARP), where the pacemaker ignores any atrial signals to prevent inappropriate re-triggering.

Transition: After the ARP duration, the system transitions to **sense** to monitor for intrinsic atrial activity.

State 3: Sense

The pacemaker monitors for natural atrial activity. If activity is detected, the pacing pulse is inhibited.

Transition: If atrial activity is detected, the state returns to **sense** to continue monitoring. If no activity is detected within the interval (LRL - ARP), the state transitions back to **Atrial_Pacing** to deliver another pacing pulse.

State 4: Delay

A timed delay state that stabilizes the interval following detected ventricular activity.

Transition: After the delay, transitions back to Sense state.

State 5: Atrial_Pacing

The pacemaker delivers an electrical pulse to the atrium, causing it to contract. After delivering the pulse, it ensures the atrial pacing circuitry is reset.

Transition: After the pulse duration, the state transitions to **AOO_Charge** again to prevent the sensing of the pacing pulse itself.

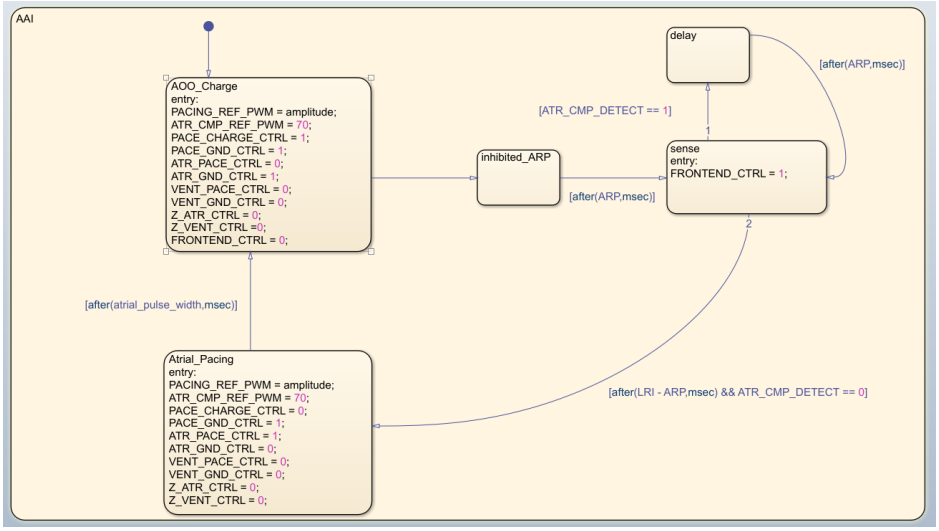


Figure 17. AAI Mode State Machine

VOO Mode State Machine

State 1: VOO_Charge

In this state, the pacemaker is preparing to deliver a pacing pulse to the ventricle. Similar to the AOO mode, the system charges the pacing circuitry and ensures the ground is properly connected to the ventricle, but it does not yet deliver the pulse. The system waits for a specific time interval, based on the pacing rate, ensuring the pacing pulse is delivered at the right moment.

Transition: After the waiting period (based on the pacing rate and pulse duration), the system moves to the pacing state to deliver the pulse.

State 2: Ventricular_Pacing

The pacemaker delivers a controlled electrical pulse to the ventricle to stimulate it to contract. Once the pulse is delivered, the system disconnects the ground and stops the pulse. This action ensures that the ventricle contracts and maintains a proper heart rhythm.

Transition: After the pulse is delivered, the system transitions back to the charging state to reset and prepare for the next pacing cycle, repeating this process continuously based on the programmed rate.

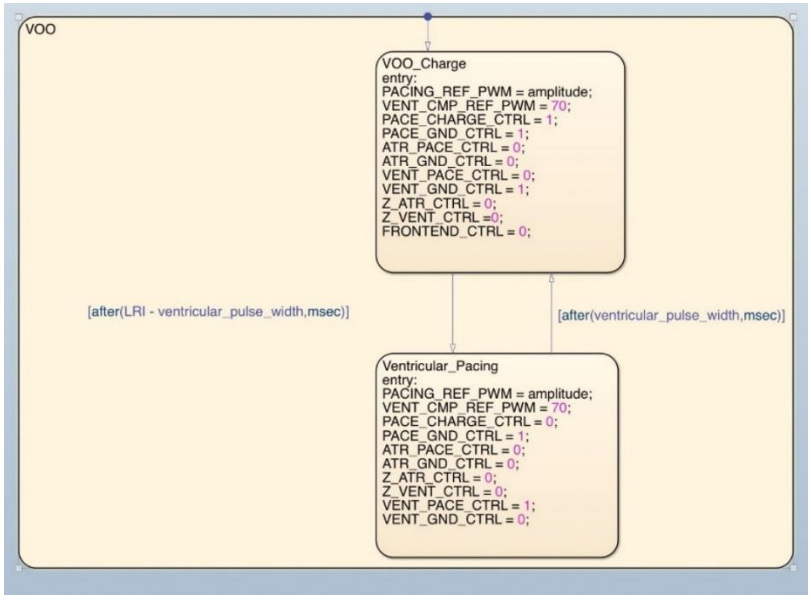


Figure 18. VOO Mode State Machine

VVI Mode State Machine

State 1: VVI_Charge

The pacemaker charges the ventricular pacing circuit and waits for the appropriate interval based on the pacing rate. During this time, it prepares to deliver a pulse if no intrinsic ventricular activity is detected.

Transition: The state transitions to **inhibited_VRP** if no intrinsic activity is detected.

State 2: inhibited_VRP

This state represents the Ventricular Refractory Period (VRP), where the pacemaker ignores any ventricular signals to prevent inappropriate re-triggering.

Transition: After the VRP duration, the system transitions to **sense** to monitor for intrinsic ventricular activity.

State 3: Sense

The pacemaker monitors for natural ventricular activity. If activity is detected, the pacing pulse is inhibited.

Transition: If ventricular activity is detected, the state returns to **sense** to continue monitoring. If no activity is detected within the interval (LRL - VRP), the state transitions to **Ventricular_Pacing** to deliver another pacing pulse.

State 4: Delay

A timed delay state that stabilizes the interval following detected ventricular activity.

Transition: After the delay, transitions back to Sense state.

State 4: Ventricular_Pacing

The pacemaker delivers an electrical pulse to the ventricle, causing it to contract. After delivering the pulse, it ensures the ventricular pacing circuitry is reset.

Transition: After the pulse duration, the state transitions back to **VVI_Charge** to prepare for the next pacing cycle.

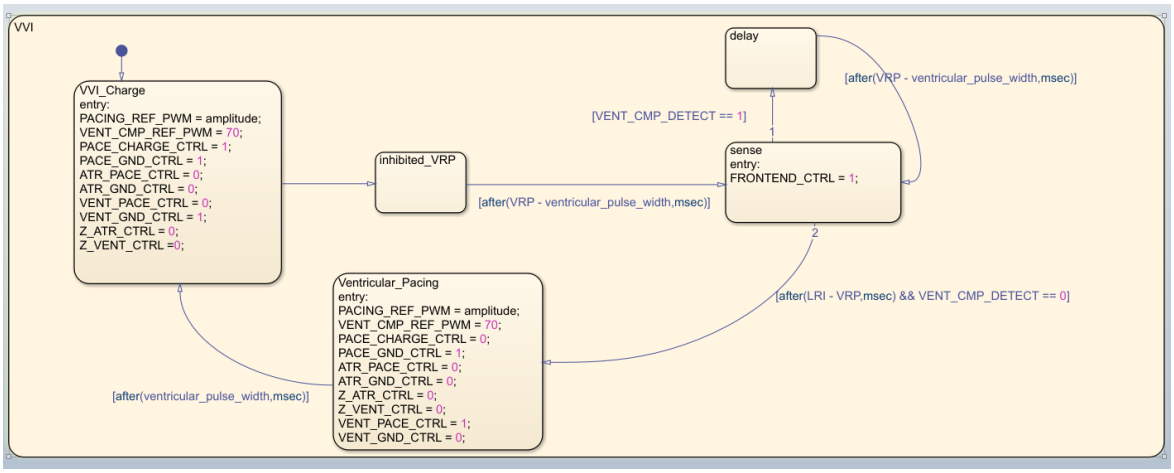


Figure 19. VVI Mode State Machine

Device-Controller Monitor

Requirements

1. User Registration Systerm

A maximum of 10 users can be supported by the Pacemaker Device Control System (DCM). In order to register, users must put in both username and password, which are saved locally. When the maximum number of users is reached, the system will enforce this user restriction by blocking additional registrations. A login procedure will authenticate users who have registered using credentials that match.

2. User Interface Design (UI)

In a controlled window system, the DCM interface (UI) must provide a user-friendly visual and interactive experience that can show both text and graphical material. In order to facilitate seamless user interaction with all DCM features, each window must enable a variety of input actions and all of the interface's panes must respond to button or value inputs.

3. Parameter and Pace Mode Management

As defined in the parameters requirment for assignmen 1. P = { Lower Rate Limit, Upper Rate Limit, Atrial Amplitude, Atrial Pulse Width, Ventricular Amplitude, Ventricular Pulse Width, VRP, ARP} and for each parameter p ϵ P, the systerm is required to provide a function or methods that can process the parameter into the specifdied pacing modes and validating the input value range. Othervise an error messag should be printed to notify the users.

Pacing Mode are specified as M = {AOO, VOO, AAI, VVI}. When pacing mode is selected, the corresponding paramter should be displayed only at a visibal position in the program. Any position that can clearly visualized by the user is applicable. For mode m ϵ M and being selected, parmeters p1 and p2 and etc should be displayed. For example:

m = M{ AOO} , p = P{ Lower Rate Limit, Upper Rate Limit, Atrial Amplitude, Atrial Pulse Width}

4. Electrogram Data Transmition

Presently, the electrogram data should be implemented as fronted display message with cabability of showing the current pacing mode and enabling the handling of electrogram data for future assignments. This data structure will efficiently capture time-series electrogram data, ensuring it is stored in a format suitable for future analysis and operations.

Design Decisions

Using Decision Matrix to display all determined design decisions.

Requirement	Design Decision	Purpose and Rationale	Additional Feature/Improvemet
Develop welcome screen for user registration and login	Implemented registration and login functionality with local storage for a maximum of 10 users	Meets the requirement of limiting user storage; simple local text storage ensures data is accessible and manageable	Deletion of registered accounts: Allows users to manage the limited storage by deleting unnecessary accounts
UI with text and graphics capability	Used PyQt's QMainWindow for window management, supporting text and graphical elements	Ensures a structured, visually consistent interface; supports the addition of images and labels for better user experience	Display username in application window: Adds personalized user experience, showing the active user
Input buttons	Implemented button-based interactions for all primary functionalities, including registration, login, etc.	Enables user interaction with the system, fulfilling functional requirements	Return key shortcut for login: Improves usability by reducing the need for extra clicks
Display of programmable parameter	Created a parameter section with QLineEdit widgets and range validation	Allows users to set and adjust required parameters while ensuring values stay within medically safe ranges	
Parameters	Added separate input fields for each programmable parameter; validated inputs with min/max range checking	Provides clear, organized parameter entry and satisfy display specified parameters for pacing mode	Pacing mode-specific parameters: Adjusts parameter visibility based on selected mode, reducing cognitive load

Device connection check	Used serial communication and visual indicators to check and display device connection status	Enables the user to see when the device is connected and communicating. Front-end design currently.	Stop telemetry button: Allows users to manually stop data transmission for safer usage and controlled data management
Communication status between DCM and device	Visual status indicator using color-coded labels	Real-time indication of connection status prevents user confusion and help in troubleshooting	
Checking PACEMAKER device model	Implemented device identity check and warning message	Provides clear indication of device identity changes, preventing unintended device interactions	
Pacing modes interface (AOO, VOO, AAI, VVI)	Created dropdown menu for pacing mode selection and adapted parameter display based on mode	Provides user flexibility and simplifies parameter adjustments. Saved space for future implementation	Real-time pacing mode display: Updates current mode display in the electrogram section, keeping users informed
Electrogram data structure	Placeholder electrogram widget; structured data storage for future integration	Meets requirement for egram display preparation; adaptable for future data visualization requirements	
Additional features	Incorporated features to enhance usability	Supports user experience, makes interaction more intuitive and secure	Exit function: Allows users to terminate the program, preventing data loss and keep stability

Validation and Verification

1. User Login/Registration/deletion functionality

Validation: The total number of users that can be registered cannot exceed 10. No action will be taken if the full name and password are not entered. No login or deletion is permitted for unregistered user names.

Verification: Error message should be printed out when any of the conditions meet.

2. Parameter Fields and Range Validation (self.fields and self.parameter_ranges)

Validation: Ensure that all of the parameters are visible as well as that the range limitations. For example, Atrial Amplitude: 0 V– 7V, Lower Rate Limit: 30 ppm–175 ppm.

Verification: Error messages appear for incorrect inputs by testing parameter entry with values both inside and outside of the permitted ranges. Verify that the field only accept numeric inputs.

3. Pacing Mode Selection and Real-Time Display

Validation: Make sure that each time users pick a pacing mode, the interface is updated to show just the parameters that are relevant to that mode.

The current pacing mode should be reflected in real-time on the electrogram display.

Verification: Verify that unnecessary fields are hidden and that the appropriate parameter fields are shown depending on the mode that is selected. Check to see if the nominal value for each parameter is seeting correctly in defalt and change according to the pacing mode.

Verify that switching between pacing modes updates the displayed mode on the electrogram label, with the correct text shown at all times.

Test and Result

User Login/Registration/deletion functionality

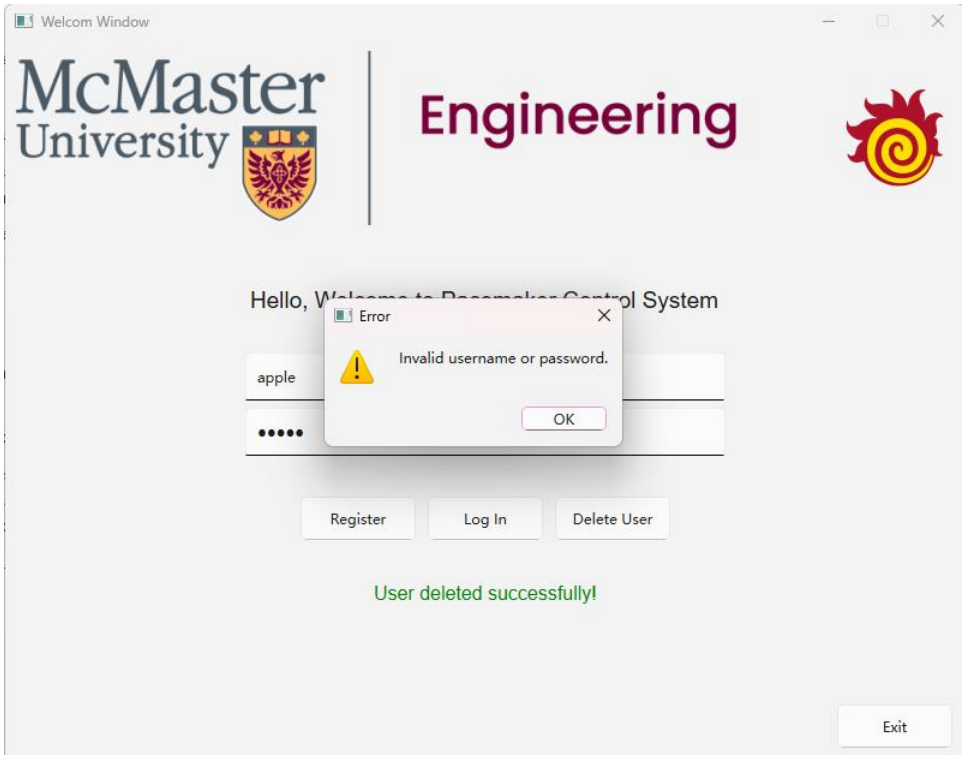


Figure 20 Testing unknown user registration

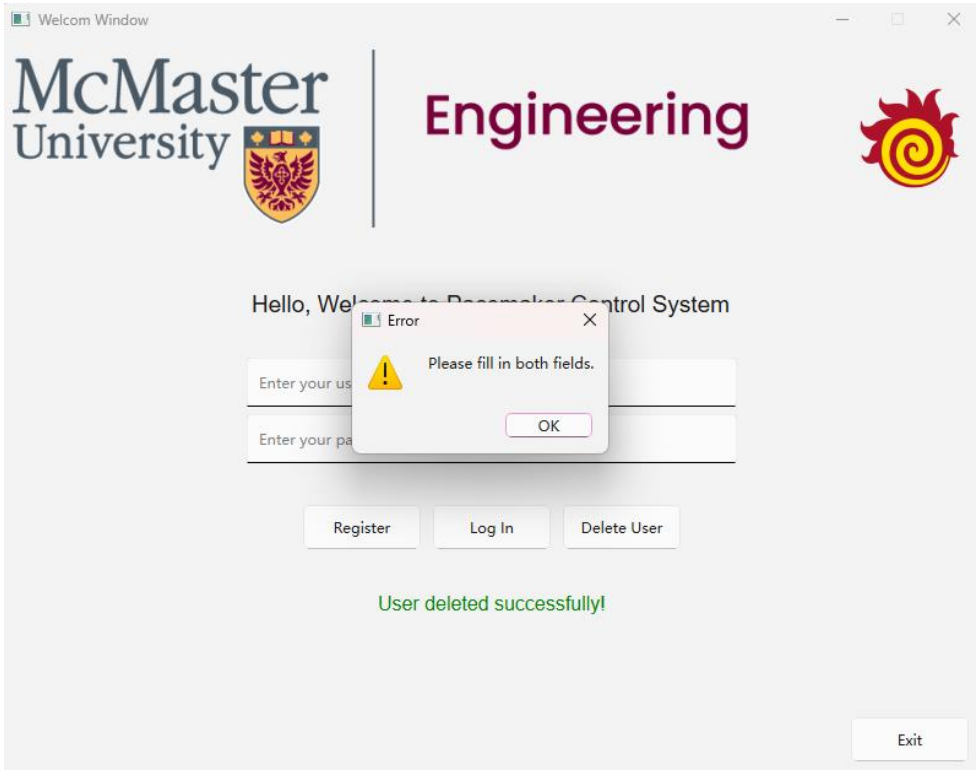


Figure 21 Testing unkown input

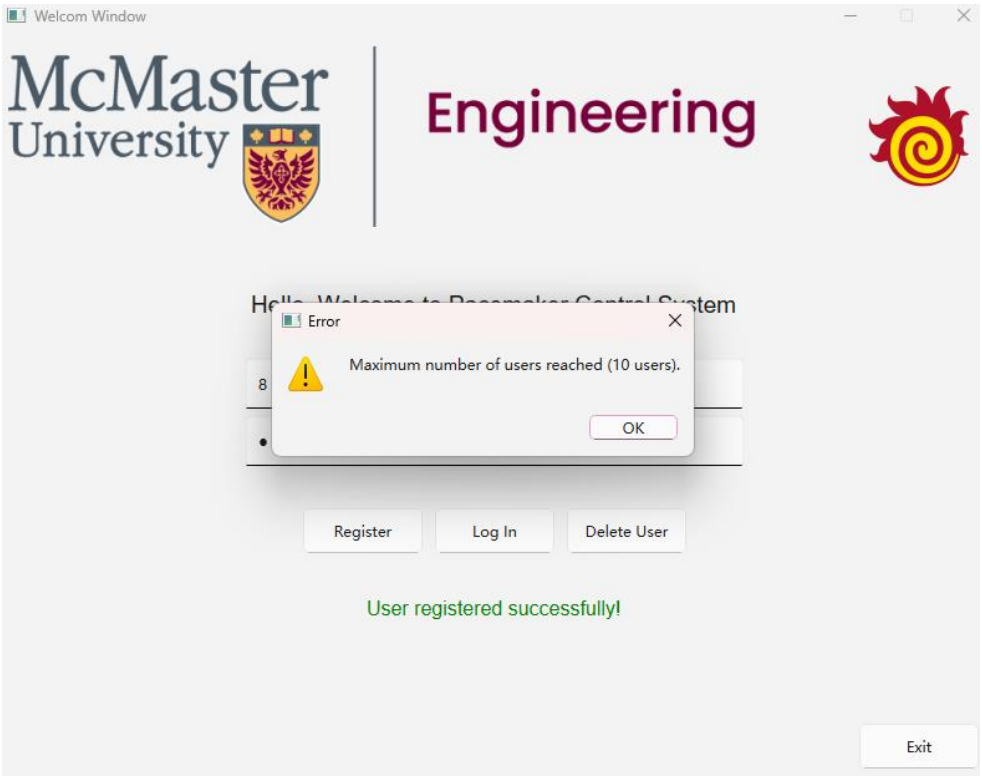


Figure 22 Testing for Maximum User

Parameter Fields and Range Validation (self.fields and self.parameter_ranges)

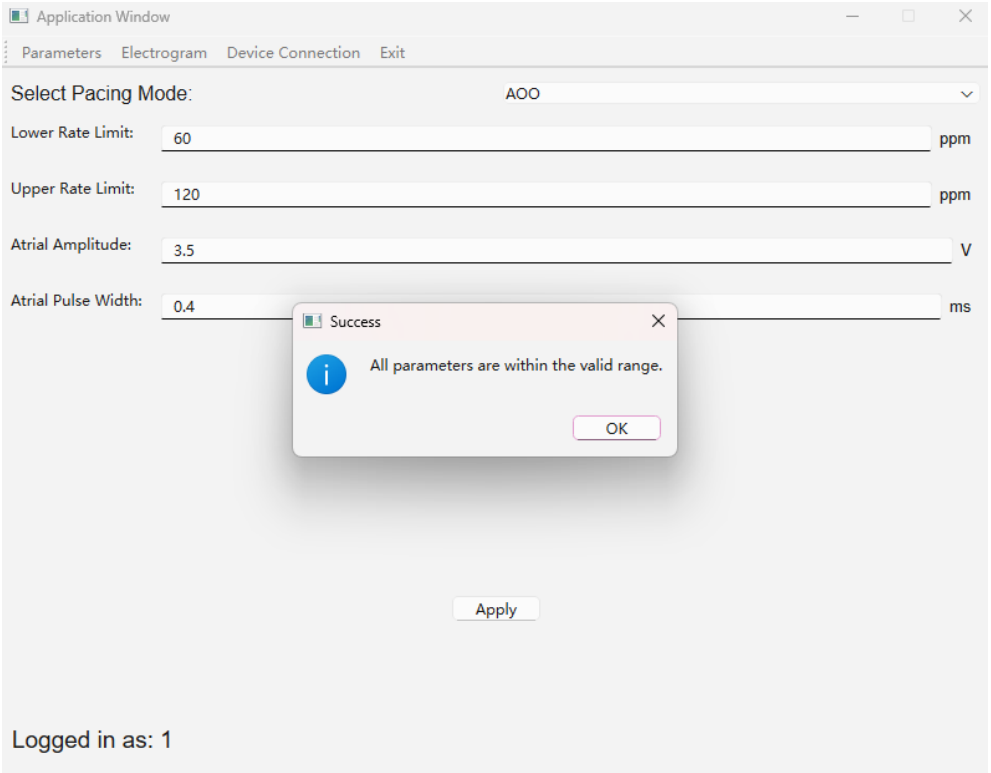


Figure 23 Testing for Valid Input of programmable parameter

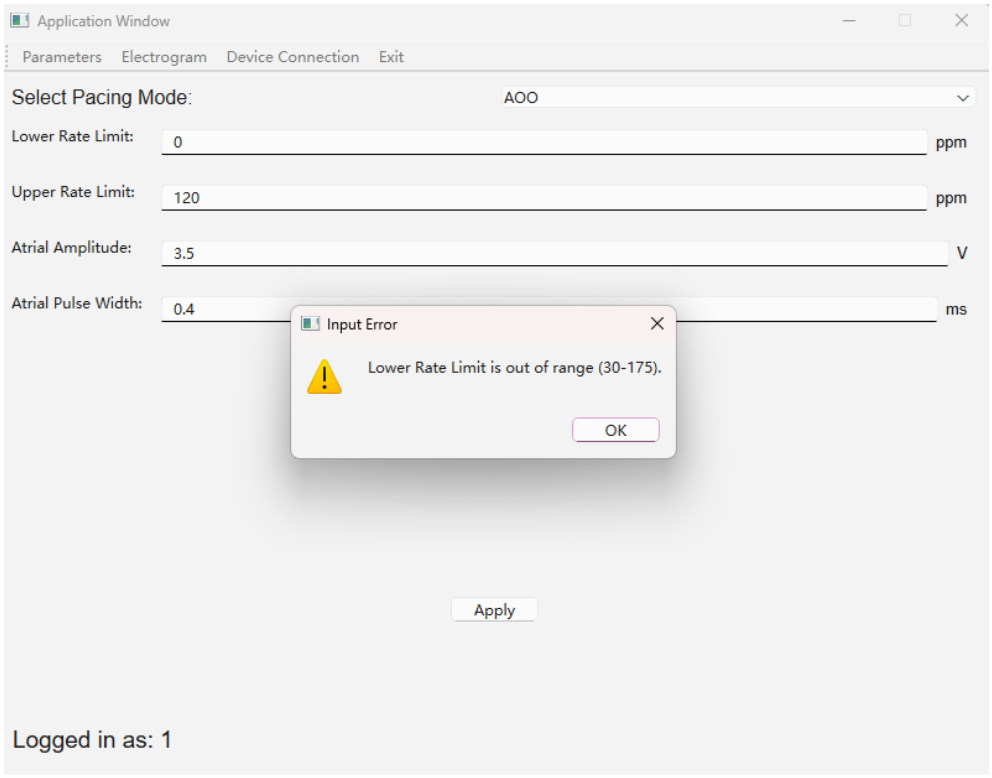


Figure 24 Testing for Invalid input parameter LRL

4. Pacing Mode Selection and Real-Time Display

The current pacing mode is selected as AOO:

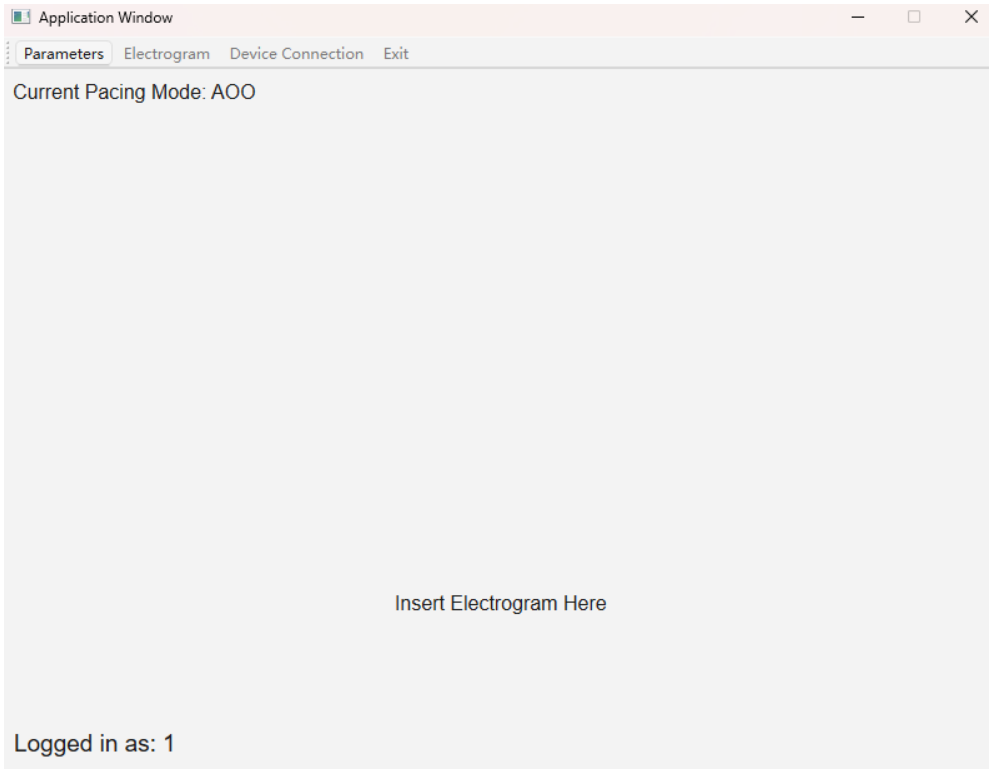


Figure 25 Testing for Electrogram Real-Time Display of Pacing Mode

Requirement Changes

1. Incorporation of Additional Pacing Modes and Parameters

Current State: The DCM currently only supports 4 pacing modes and 8 parameters.

Expected Change: Future assignments will require additional pacing modes and parametrs listed on the entire project requirments. New controls and data validation processes may be necessary to handle additional parameters related to the new pacing modes. Backend code will need expansion to incorporate the logic and validation for these new pacing parameters.

2. Real-Time Electrogram Data Visualization

Current State: Currently, there is a placeholder for electrogram (EGM) data display without real-time data processing.

Expected Change: Future requirements specify real-time electrogram data visualization, possibly including markers for pacing events, heart activity, and other time-series data.

3. Data Storage and Retrieval of Parameter Settings

Current State: The current method of storing parameters are not designed as it only determine the value range.

Expected Change: Future assignments may require persistent storage of programmable parameters for continuity across sessions. A more robust data storage mechanism like SQLite or JSON-based data storage might be a good choice.

4. Integration of Serial Communication Protocols

Current State: The device connection feature is currently simulated, with limited interaction with serial devices.

Expected Change: Future requirements will include testing and harware connecting with the board.

5. User Interface Enhancements for Further Need

Current State: The current UI is functional but lacks specific accessibility features (e.g., support for color-blind users or screen reader compatibility).

Expected Change: Future development may require enhanced accessibility options, especially the device is used by patients directly. The possible option could include a voice system or icon-marked buttons to help the visually or hearing challenged group.

Design Decision Changes

UI with Text and Graphics:

Expand the UI to support additional elements like real-time electrogram visualization, accessibility features (e.g., high-contrast modes), and improved layout adaptability.

Input Buttons:

Refine button functionality to accommodate future tasks, such as data print out setting to complete a pacemaker functionalities.

Parameter Display and Validation:

Adapt validation logic to handle additional pacing modes and more complex pacing state sets.

Device Connection and Communication:

Implement communication protocal and test for a stable connection. Add different device checking procedure.

Pacing Mode Interface:

Add options for future pacing modes while ensuring that new parameters can be displayed seamlessly without major UI changes.

Electrogram Data Structure:

Replace the placeholder with a real-time data visualization module, including data input and output connection pins with the actual hardware.

DCM Code

1. main_Window.py

```
import sys
from PyQt6.QtCore import *
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
from application_window import ApplicationWindow

# Main Class to customize pacemaker's welcome window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # Set the window title
        self.setWindowTitle("Welcom Window")

        # Set the dimensions of the main window
        self.setFixedSize(QSize(800, 600))

        # Load images and set size
        mac_img = QPixmap("mac.png").scaled(600, 200,
Qt.AspectRatioMode.KeepAspectRatio,
Qt.TransformationMode.SmoothTransformation)
        fireball_img = QPixmap("fireball.png").scaled(100, 100,
Qt.AspectRatioMode.KeepAspectRatio,
Qt.TransformationMode.SmoothTransformation)

        # Create QLabel for welcome msg
        msg = QLabel('Hello, Welcome to Pacemaker Control System')
        msg.setFont(QFont('Arial', 14))

        # Create QLabel widgets to display the images
        mac_label = QLabel()
        mac_label.setPixmap(mac_img if not mac_img.isNull() else
QPixmap())

        fireball_label = QLabel()
        fireball_label.setPixmap(fireball_img if not
fireball_img.isNull() else QPixmap())

        # Create user registration, login, and delete existing user input
line
        self.username_input = QLineEdit()
        self.username_input.setFixedSize(400, 40)
        self.username_input.setPlaceholderText("Enter your username")

        self.password_input = QLineEdit()
        self.password_input.setFixedSize(400, 40)
        self.password_input.setPlaceholderText("Enter your password")
        self.password_input.setEchoMode(QLineEdit.EchoMode.Password)

        # Create buttons:
        # registration
        self.register = QPushButton("Register")
        self.register.setFixedSize(100, 40)
        self.register.clicked.connect(self.register_user)
```

Figure 26 DCM Code for Class MainWindow

```
        # login
        self.login = QPushButton("Log In")
        self.login.setFixedSize(100, 40)
        self.login.clicked.connect(self.login_user)
        # Delete users
        self.delete = QPushButton("Delete User")
        self.delete.setFixedSize(100, 40)
        self.delete.clicked.connect(self.delete_user)
        # Exit button
        self.exit = QPushButton("Exit")
        self.exit.setFixedSize(100, 40)
        self.exit.clicked.connect(self.exit_program)

        # Connect the Enter key to log in button
        shortcut = QShortcut(QKeySequence(Qt.Key.Key_Return), self)  #
Key_Return refers to enter key
        shortcut.activated.connect(self.login.click)

        # Create a layout and add the widgets
        main_layout = QVBoxLayout()  # Vertical layout

        # Create a top and horizontal layout to hold images
        img_layout = QHBoxLayout()
        img_layout.addWidget(mac_label,
alignment=Qt.AlignmentFlag.AlignLeft)
        img_layout.addWidget(fireball_label,
alignment=Qt.AlignmentFlag.AlignRight)

        # Create a horizontal layout for the buttons
        button_layout = QHBoxLayout()
        button_layout.addWidget(self.register,
alignment=Qt.AlignmentFlag.AlignRight | Qt.AlignmentFlag.AlignTop)
        button_layout.addWidget(self.login,
alignment=Qt.AlignmentFlag.AlignVCenter | Qt.AlignmentFlag.AlignTop)
        button_layout.addWidget(self.delete,
alignment=Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop)

        # Success message QLabel
        self.success_msg = QLabel("")
        self.success_msg.setFont(QFont('Arial', 12))
        self.success_msg.setStyleSheet("color: green;")  # Set the color
to green for success messages

        # Arrange layout vertical order and combine them together
        main_layout.addLayout(img_layout)
        main_layout.addWidget(msg,
alignment=Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignBottom)
        main_layout.addWidget(self.username_input,
alignment=Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignBottom)
        main_layout.addWidget(self.password_input,
alignment=Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignTop)
        main_layout.addLayout(button_layout)
        main_layout.addWidget(self.success_msg,
alignment=Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignTop)
```

Figure 27 DCM Code for Class MainWindow

```
        main_layout.addWidget(self.exit,
alignment=Qt.AlignmentFlag.AlignBottom | Qt.AlignmentFlag.AlignRight)

        # Set the layout for the central widget
        central_widget = QWidget()
        central_widget.setLayout(main_layout)
        self.setCentralWidget(central_widget)

    def register_user(self):
        # Get username and password
        username = self.username_input.text()
        password = self.password_input.text()

        if username and password:
            # Check if the maximum number of users has been reached
            try:
                with open('users.txt', 'r') as file:
                    lines = file.readlines()

                if len(lines) >= 10:
                    QMessageBox.warning(self, "Error", "Maximum number of
users reached (10 users).")
                    return
            except FileNotFoundError:
                # File doesn't exist yet, so no users are registered
                pass

            # Store the credentials in a text file
            with open('users.txt', 'a') as file:
                file.write(f"{username},{password}\n")

            # Display success message
            self.success_msg.setText("User registered successfully!")

            # Clear input fields
            self.username_input.clear()
            self.password_input.clear()
        else:
            # Display error message if fields are not filled
            QMessageBox.warning(self, "Error", "Please fill in both
fields.")

    def delete_user(self):
        # Get username and password
        username = self.username_input.text()
        password = self.password_input.text()

        if username and password:
            # Read the existing users from the file
            try:
                with open('users.txt', 'r') as file:
                    lines = file.readlines()

                # Write back all users except the one to be deleted
```

Figure 28 DCM Code for Class MainWindow

```
                with open('users.txt', 'w') as file:
                    user_found = False
                    for line in lines:
                        stored_username, stored_password =
line.strip().split(',')
                        if stored_username == username and
stored_password == password:
                            user_found = True
                            continue  # Skip writing this user to the
file
                        file.write(line)

                if user_found:
                    self.success_msg.setText("User deleted
successfully!")
                else:
                    QMessageBox.warning(self, "Error", "User not found.")
            except FileNotFoundError:
                QMessageBox.warning(self, "Error", "No users registered
yet.")
        else:
            # Display error message if fields are not filled
            QMessageBox.warning(self, "Error", "Please fill in both
fields.")

    def login_user(self):
        # Get username and password
        username = self.username_input.text()
        password = self.password_input.text()

        if username and password:
            try:
                with open('users.txt', 'r') as file:
                    lines = file.readlines()

                # Check if user credentials match any registered user
                for line in lines:
                    stored_username, stored_password =
line.strip().split(',')
                    if stored_username == username and stored_password ==
password:
                        self.success_msg.setText("Login successful!")
                        self.open_application_window(username)
                        return

                QMessageBox.warning(self, "Error", "Invalid username or
password.")
            except FileNotFoundError:
                QMessageBox.warning(self, "Error", "No users registered
yet.")
        else:
            QMessageBox.warning(self, "Error", "Please fill in both
fields.")
```

Figure 29 DCM Code for Class MainWindow

```
    def open_application_window(self, username):
        # Don't hide the log in window so more user can be logged in
        # Open the main application window and pass the username
        self.app_window = ApplicationWindow(username)
        self.app_window.show()

    def exit_program(self):
        QApplication.quit()

# Main function to run the application
def main():
    app = QApplication(sys.argv) # Create an instance of QApplication
    window = MainWindow()
    window.show()  # Show the main window
    sys.exit(app.exec()) # Start the application's event loop

if __name__ == "__main__":
    main()
```

Figure 30 DCM Code for Class MainWindow

2. application_Window.py

```
import serial
import serial.tools.list_ports
from PyQt6.QtCore import *
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *

# Sub Class to customize pacemaker's application window
class ApplicationWindow(QMainWindow):
    def __init__(self, username):
        super().__init__()

        # Section: Initialization
        # --------------------------------
        # Parameter fields
        self.fields = {
            "Lower Rate Limit": QLineEdit(),
            "Upper Rate Limit": QLineEdit(),
            "Atrial Amplitude": QLineEdit(),
            "Atrial Pulse Width": QLineEdit(),
            "Ventricular Amplitude": QLineEdit(),
            "Ventricular Pulse Width": QLineEdit(),
            "VRP": QLineEdit(),
            "ARP": QLineEdit()
        }
        self.current_pacing_mode = None  # Set initial pacing mode as
none

        # Parameter Ranges
        self.parameter_ranges = {
            "Lower Rate Limit": (30, 175),
            "Upper Rate Limit": (50, 175),
            "Atrial Amplitude": (0, 7),
            "Atrial Pulse Width": (0.05, 1.9),
            "Ventricular Amplitude": (0, 7),
            "Ventricular Pulse Width": (0.05, 1.9),
            "VRP": (150, 500),
            "ARP": (150, 500)
        }

        # Window title and dimensions
        self.setWindowTitle("Application Window")
        self.setFixedSize(QSize(800, 600))

        # Display the logged-in user's name
        self.username_label = QLabel(f"Logged in as: {username}", self)
        self.username_label.setFont(QFont('Arial', 14))
        self.username_label.setGeometry(10, 550, 300, 30)  # Set position
and size of label

        # Section: Central Stacked Widget
        # --------------------------------
        # Create a central stacked widget to hold different views
        self.central_stack = QStackedWidget()
        self.setCentralWidget(self.central_stack)
```

Figure 31 DCM Code for Class ApplicationWindow

```
        # Create individual widgets for each functionality
        self.parameters_widget = self.create_parameters_widget()
        self.egram_widget = self.create_egram_widget()
        self.connection_widget = self.create_connection()

        # Add each widget to the central stacked widget
        self.central_stack.addWidget(self.parameters_widget)
        self.central_stack.addWidget(self.egram_widget)
        self.central_stack.addWidget(self.connection_widget)

        # Section: Toolbar
        # --------------------------------
        # Create Toolbar
        toolbar = QToolBar("Main Toolbar")
        self.addToolBar(toolbar)

        # Add actions to the toolbar
        parameters_action = QAction("Parameters", self)
        parameters_action.triggered.connect(lambda:
self.central_stack.setCurrentWidget(self.parameters_widget))
        toolbar.addAction(parameters_action)

        egram_action = QAction("Electrogram", self)
        egram_action.triggered.connect(lambda:
self.central_stack.setCurrentWidget(self.egram_widget))
        toolbar.addAction(egram_action)

        connection_action = QAction("Device Connection", self)
        connection_action.triggered.connect(lambda:
self.central_stack.setCurrentWidget(self.connection_widget))
        toolbar.addAction(connection_action)

        # Add Exit button to the toolbar
        exit_action = QAction("Exit", self)
        exit_action.triggered.connect(self.exit)
        toolbar.addAction(exit_action)

    # Section: Parameters Widget
    # --------------------------------
    def create_parameters_widget(self):
        # Create a widget for setting programmable parameters
        widget = QWidget()
        layout = QVBoxLayout()

        # Create a layout for the pacing mode selection at the top
        pacing_mode_layout = QHBoxLayout()
        pacing_mode_label = QLabel("Select Pacing Mode:")
        pacing_mode_label.setFont(QFont('Arial', 12))

        # Dropdown menu for pacing modes
        self.pacing_mode_combo = QComboBox()
        self.pacing_mode_combo.addItems(["None", "AOO", "VOO", "AAI",
"VVI"])  # Added 'None' as the first option
```

Figure 32 DCM Code for Class ApplicationWindow

```
self.pacing_mode_combo.currentIndexChanged.connect(self.update_parameters
)

        # Add label and drop down menu to the layout
        pacing_mode_layout.addWidget(pacing_mode_label)
        pacing_mode_layout.addWidget(self.pacing_mode_combo)

        # Create a form layout for parameter fields
        self.param_form_layout = QFormLayout()
        layout.addLayout(pacing_mode_layout)
        layout.addLayout(self.param_form_layout)

        # Units for each parameter
        units = {
            "Lower Rate Limit": "ppm",
            "Upper Rate Limit": "ppm",
            "Atrial Amplitude": "V",
            "Atrial Pulse Width": "ms",
            "Ventricular Amplitude": "V",
            "Ventricular Pulse Width": "ms",
            "VRP": "ms",
            "ARP": "ms"
        }

        # Create dictionaries to hold field containers and labels
        self.field_containers = {}
        self.field_labels = {}

        # Add fields to the form layout with units
        for field_name, field_widget in self.fields.items():
            # Create a horizontal layout to hold the field and the unit
label
            field_layout = QHBoxLayout()
            field_layout.addWidget(field_widget)

            # Add unit label if defined for the field
            unit_label = QLabel(units.get(field_name, ""))
            unit_label.setFont(QFont('Arial', 10))
            field_layout.addWidget(unit_label)

            # Create a container widget for the field and unit
            container = QWidget()
            container.setLayout(field_layout)

            # Store the container and label
            self.field_containers[field_name] = container
            label = QLabel(f"{field_name}:")
            self.field_labels[field_name] = label

            # Add the label and container to the form layout
            self.param_form_layout.addRow(label, container)

        # Add Apply Button to validate input
```

Figure 33 DCM Code for Class ApplicationWindow

```
        self.apply_button = QPushButton("Apply")
        self.apply_button.clicked.connect(self.validate_parameters)
        layout.addWidget(self.apply_button,
alignment=Qt.AlignmentFlag.AlignCenter)

        widget.setLayout(layout)
        return widget

    # Function to valiadate whether the input value is out of range
    def validate_parameters(self):
        for field_name, (min, max) in self.parameter_ranges.items():
            field_text = self.fields[field_name].text()
            if field_text:  # Only check if there is an input
                try:
                    value = float(field_text)
                    if not (min <= value <= max):
                        QMessageBox.warning(self, "Input Error",
f"{field_name} is out of range ({min}-{max}).")
                        return  # Exit on the first error
                except ValueError:
                    QMessageBox.warning(self, "Input Error", f"Invalid
input for {field_name}. Please enter a numeric value.")
                    return

        QMessageBox.information(self, "Success", "All parameters are
within the valid range.")


    def update_parameters(self):
        # Define the relevant parameters to show for each mode
        mode = self.pacing_mode_combo.currentText()
        self.current_pacing_mode = mode

        self.update_egram_label()  # Update the Egram label with the
current pacing mode

        # Define specified parameters for each pacing mode
        if mode == "AOO":
            self.set_field_visibility({
                "Lower Rate Limit": True,
                "Upper Rate Limit": True,
                "Atrial Amplitude": True,
                "Atrial Pulse Width": True,
                "Ventricular Amplitude": False,
                "Ventricular Pulse Width": False,
                "VRP": False,
                "ARP": False
            })
            self.set_nominal_value("60", "120", "3.5", "0.4", "", "", "",
"")

        elif mode == "AAI":
            self.set_field_visibility({
                "Lower Rate Limit": True,
```

Figure 34 DCM Code for Class ApplicationWindow

```
                "Upper Rate Limit": True,
                "Atrial Amplitude": True,
                "Atrial Pulse Width": True,
                "Ventricular Amplitude": False,
                "Ventricular Pulse Width": False,
                "VRP": True,
                "ARP": True
            })
            self.set_nominal_value("60", "120", "3.5", "0.4", "", "",
"320", "250")

        elif mode == "VOO":
            self.set_field_visibility({
                "Lower Rate Limit": True,
                "Upper Rate Limit": True,
                "Atrial Amplitude": False,
                "Atrial Pulse Width": False,
                "Ventricular Amplitude": True,
                "Ventricular Pulse Width": True,
                "VRP": False,
                "ARP": False
            })
            self.set_nominal_value("60", "120", "", "", "3.5", "0.4", "",
"")

        elif mode == "VVI":
            self.set_field_visibility({
                "Lower Rate Limit": True,
                "Upper Rate Limit": True,
                "Atrial Amplitude": False,
                "Atrial Pulse Width": False,
                "Ventricular Amplitude": True,
                "Ventricular Pulse Width": True,
                "VRP": True,
                "ARP": False
            })
            self.set_nominal_value("60", "120", "", "", "3.5", "0.4",
"320", "")

        else:
            self.clear_parameters()

    def set_field_visibility(self, visibility_dict):
        # Set visibility for each parameter field container and label
        for field_name, is_visible in visibility_dict.items():
            container = self.field_containers.get(field_name)
            label = self.field_labels.get(field_name)
            if container:
                container.setVisible(is_visible)
            if label:
                label.setVisible(is_visible)
```

Figure 35 DCM Code for Class ApplicationWindow

```
    def set_nominal_value(self, lower_rate_limit, upper_rate_limit,
atrial_amplitude, atrial_pulse_width, ventricular_amplitude,
ventricular_pulse_width, vrp, arp):
        # Set parameter values in the QLineEdit widgets only for visible
fields
        if self.fields["Lower Rate Limit"].isVisible():
            self.fields["Lower Rate Limit"].setText(lower_rate_limit)
        if self.fields["Upper Rate Limit"].isVisible():
            self.fields["Upper Rate Limit"].setText(upper_rate_limit)
        if self.fields["Atrial Amplitude"].isVisible():
            self.fields["Atrial Amplitude"].setText(atrial_amplitude)
        if self.fields["Atrial Pulse Width"].isVisible():
            self.fields["Atrial Pulse Width"].setText(atrial_pulse_width)
        if self.fields["Ventricular Amplitude"].isVisible():
            self.fields["Ventricular
Amplitude"].setText(ventricular_amplitude)
        if self.fields["Ventricular Pulse Width"].isVisible():
            self.fields["Ventricular Pulse
Width"].setText(ventricular_pulse_width)
        if self.fields["VRP"].isVisible():
            self.fields["VRP"].setText(vrp)
        if self.fields["ARP"].isVisible():
            self.fields["ARP"].setText(arp)

    def clear_parameters(self):
        # Clears all fields in the form layout
        for field_widget in self.fields.values():
            field_widget.clear()

    def update_egram_label(self):
        # Update electrogram widget label with the current pacing mode
        if hasattr(self, 'egram_label'):
            self.egram_label.setText(f"Current Pacing Mode:
{self.current_pacing_mode}")

    # Section: Electrogram Widget
    # --------------------------------
    def create_egram_widget(self):
        # Create a widget for real-time electrogram data (Placeholder)
        widget = QWidget()
        layout = QVBoxLayout()

        # Add a label to display the current pacing mode
        self.egram_label = QLabel("Current Pacing Mode: None")
        self.egram_label.setFont(QFont('Arial', 12))
        layout.addWidget(self.egram_label,
alignment=Qt.AlignmentFlag.AlignTop | Qt.AlignmentFlag.AlignLeft)

        # Add the egram graph in the future
        self.egram_graph = QLabel("Insert Electrogram Here")
        self.egram_graph.setFont(QFont('Arial', 12))
        layout.addWidget(self.egram_graph,
alignment=Qt.AlignmentFlag.AlignCenter)
```

Figure 36 DCM Code for Class ApplicationWindow

```
        widget.setLayout(layout)
        return widget

    # Section: Device Connect Status Widget
    # ---------------------------------
    def create_connection(self):
        widget = QWidget()
        layout = QVBoxLayout()

        # Create buttons
        self.connect_button = QPushButton("Connection Status")
        self.disconnect_button = QPushButton("Disconnect Telemetry")

        # Set button sizes
        self.connect_button.setFixedSize(300, 60)
        self.disconnect_button.setFixedSize(300, 60)

        # Create a single status message label
        self.status_msg = QLabel("Device not connected")
        self.status_msg.setFont(QFont('Arial', 12))
        self.status_msg.setStyleSheet("color: red;")
        layout.addWidget(self.status_msg,
alignment=Qt.AlignmentFlag.AlignCenter)

        # Button layout
        layout.addWidget(self.connect_button,
alignment=Qt.AlignmentFlag.AlignCenter | Qt.AlignmentFlag.AlignBottom)
        layout.addWidget(self.disconnect_button,
alignment=Qt.AlignmentFlag.AlignCenter | Qt.AlignmentFlag.AlignTop)

        # Connect the buttons to the appropriate slots
        self.connect_button.clicked.connect(self.check_connection_status)
        self.disconnect_button.clicked.connect(self.stop_telemetry)

        widget.setLayout(layout)
        return widget

    def check_connection_status(self):
        # Check available serial ports
        ports = serial.tools.list_ports.comports()
        for port in ports:
            # Pacemaker ID (Not tested)
            if "H00140" in port.description:
                self.status_msg.setText(f"Connected device:
{port.device}")
                self.status_msg.setStyleSheet("color: green;")
                return

        # If no device is found
        self.status_msg.setText(f"Connected device: None")
        self.status_msg.setText("Device not connected")
        self.status_msg.setStyleSheet("color: red;")

    def stop_telemetry(self):
```

Figure 37 DCM Code for Class ApplicationWindow

```
    # Set the logic here when actually connecting with the device
    # A condition to detect that the telemetry is on
    self.status_msg.setText("Telemetry is stopped")
    self.status_msg.setStyleSheet("color: red;")

# Section: Program exit
# ---------------------------------
def exit(self):
    self.hide()
```

Figure 38 DCM Code for Class ApplicationWindow

Module Clarification

1. Main_Window.py

Purpose: Provides a GUI that manage different users, allowing them to register an account (up to 10), login, and even delete a pre-existing account. Upon logging in, they will be re-directed to the Application Window

Public Functions:

A. __init__(self)

- Initializes main window, user input fields, buttons, labels, and layouts
- Parameters: None

B. register_user(self)

- Parameters: None

C. delete_user(self)

- Parameters: None

D. login_user(self)

- Parameters: None

E. open_application_window(self, username)

- Parameters: username (string) - username of currently logged-in user

F. Exit_program(self)

- Parameters: None

Black-Box Behaviour of Each Function:

A. __init__(self): sets up UI elements. Window is displayed with the components for user input and buttons. Does not return a value.

B. register_user(self): validates input fields. If valid, adds a new usesr to a file and displays a success message. If invalid, shows an error message

C. delete_user(self): validates input fields. If valid, removes a user from a file and displays a success message or an error if user not found

D. login_user(self): validates input fields. If valid, checks inputs against a file. If correct, opens the application window. If not, shows an error message.

E. Open_application_window(self, username): opens ApplicationWindow with username of logged-in user

F. Exit_program(self): closes application

Global Variables (state variables): None. The state of the application is managed through instance variables set in __init__.

Private Functions: No private functions in this module. All can be considered public.

Internal Behaviour of Each Public Function:

- __init__(self)
  - Initializes UI components and layouts
  - Sets up signal-slot-connection for buttons and keyboard shortcuts
  - Displays main window with layouts
- Register_user(self)
  - Retrieves username and password from input fields
  - Checks if both fields are filled
  - Opens user.txt in read mode to check number of users. If 10+, shows warning
  - If valid, opens users.txt in append mode and writes the new user details

    - Updates success message label and clears input fields
- Deletes_user(self)
    - Retrieves username and password
    - Checks if both fields filled
    - Opens users.txt in read mode to read existing users
    - Writes back all users except one to be deleted
    - Updates message based on if user found
- Login_user(self)
    - Retrieves username and password
    - Checks if both fields filled
    - Opens users.txt in read mode to validate
    - If found, sets sucess message and calls open_application_window();
    - If not, shows error
- Open_application_window(self, username)
    - Creates instance of application window, passes username and shows it
- Exit_program(self)
    - Calls QApplication.quit() to exit

2. Application_Window.py

Purpose: provides a GUI for users to manage the parameters of the connected pacemaker device. It allows the users to select pacing modes, input parameters for specific modes, validate those inputs based on the set intervals, check the connection status of the device, and even exit the window (doing this will take you back to the Welcome Window).

Public Functions:

A. __init__(self, username)

- Parameters: username (string) - name of the user logged into the application

B. create_parameters_widget(self)

- Parameters: None

C. validate_parameters(self)

- Parameters: None

D. update_parameters(self)

- Parameters: None

E. set_field_visibility(self, visibility_dict)

- Parameters: visibility_dict (dict) - sets visibility for each parameter field container and label

F. set_nominal_values(self, lower_rate_limit, upper_rate_limit, atrial_amplitude, atrial_pulse_width, ventricular_amplitude, ventricular_pulse_width, vrp, arp)

- Parameters: all parameter values are strings

G. clear_parameters(self)

- Parameters: None

H. udpate_egram_label(self)

- Parameters: None

I. create_egram_widget(self)

- Parameters: None

J. create_connection(self)

- Parameters: None

K. check_connection_status(self)

- Parameters: None

L. stop_telemetry(self)

- Parameters: None

M. exit(self)

- Parameters: None

Black-Box Behaviour of Each Function:

A. __init__: initializes application window, sets up GUI elements, and creates parameter fields and a toolbar

B. create_parameters_widget: creates a widget for setting programmable parameters. Includes a drop-down menu to select the pacing mode and input fields with the pre-set units

C. validate_parameters: validates whether the input value is out of range. If out of range, will display an error message.

D. update_parameters: updates visible input fields and nominal values based on the selected pacing mode in the combo box

E. set_field_visibility: sets visibility for each parameter field container and label (each pacing mode)

F. set_nominal_value: sets default parameter values for visible fields based on pacing mode

G. clear_parameters: clears all input fields

H. update_egram_label: updates electrogram widget label with the current pacing mode

I. create_egram_widget: creates a widget for displaying real-time electrogram data

J. create_connection: creates a widget to display connection status with buttons to check and disconnect telemetry

K. check_connection_status: checks if any device is connected to the serial ports. Displays a status message based on current connection status.

L. stop_telemetry: detects if the telemetry is stopped, updates status message if telemetry is stopped.

M. exit: hides the application window, basically exiting it.

Global Variables (State Variables):

A. self.fields: contains QLineEdit widgets for various parameters, allowing for dynamic access to input fields throughout the class.

B. self.current_pacing_mode: a string variable that stores the current pacing mode. Influences parameter visibility and default values.

C. self.parameter_ranges: defines each parameter's valid range, aiding in input validation.

Private Functions: There are no private functions in this module, all functions are public.

Internal Behaviour of Each Function:

A. __init__:

- Initializes parameter fields (self.fields) which contains QLineEdit widgets for various multiple parameters (Lower Rate Limit, Upper Rate Limit, Atrial Amplitude, Atrial Pulse Width, Ventricular Amplitude, Ventricular Pulse Width, VRP, ARP).
- Initializes the current pacing mode (self.current_pacing_mode), a variable initialized to None to track the currently selected pacing mode.
- Initializes parameter ranges (self.parameter_ranges): contains the valid ranges for each parameter: Lower Rate Limit = 30 – 175, Upper Rate Limit = 50 – 175, Atrial Amplitude = 0 – 7, Atrial pulse Width = 0.05 - 1.9, Ventricular Amplitude = 0 –7, Ventricular Pulse Width = 0.05 - 1.9, VRP = 150 – 500, ARP = 150 – 500
- Initializes window title ("application window") and dimensions (800x600 pixels)
- Initializes username label (self.username_label) - a QLabel displaying the currently user's name.
- Initializes the central stacked widget (self.central_stack) - a Qstackedwidget that holds different views for parameters, electrograms, and connection status.
- Initializes widgets for each functionality: self.parameters_widget – created by calling self.create_parameters_widget(), self.egram_widget: created by calling self.create_egram_widget(), self.connection_widget: created by calling self.create_connection().
- Initializes toolbar (toolbal) - a QToolBar added to the main window.
- Initializes toolbar actions: parameters_action – action for switching to parameters view, egram_action – action for switching to egram view,

connection_action – action for switching to connection status view, exit_action – action for exiting application

B. create_parameters_widget:

- Creates a vertical layout that includes a dropdown for pacing modes and input fields with units
- Sets up the layout and adds an “Apply” button to validate inputs

C. validate_parameters:

- loops through each field, converts input to float, and checks against defined range. If out of range or invalid, shows error message.
- Validates contents of self.fields to maintain its state

D. update_parameters:

- Retrieves selected pacing mode and adjusts visibility of input fields. Sets nominal values for parameters based on the mode.
- Modifies self.current_pacing_mode

E. set_field_visibility:

- Takes a dictionary and updates each field’s visibility based on that
- This function alerts the state of the GUI dynamically

F. set_nominal_value:

- Sets default values for visible parameter fields. It checks visibility before setting values so as to ensure only the necessary fields are updated

G. Clear_parameters:

- Iterates over all fields and clears their content, ensuring no residual data effects future inputs

H. update_egram_label:

- Updates electrogram widget’s label based on current pacing mode, maintaining dynamic connection to user input

I. create_egram_widget:

- Constructs a widget for electrogram display

J. create_connection:

- creates button sizes and layouts, and status message labels, while also connecting the buttons to the apppropriate slots

K. check_connection_status:

- Scans serial ports, updates connection status based on whether the pacemaker device is detected, and maintains state by updating self.status_msg

L. stop_telemetry:

- Updates status message indicating telemetry is inactive

M. exit:

- Hides application window, signaling and end to the program’s execution.