SFWR ENG 3K04 ASSIGNMENT 1&2

SFWR ENG 3K04 – Software Development

Professor: Thomas Chiang

Assignment 1 Date of Submission: October 25, 2024

Assignment 2 Date of Submission: November 29, 2024

Team: Wednesday_The Redliners

John Sarga_ Sargaj

Jiayi Yang (Max)_yangj285

Zihao Zhou (Bill)_ zhouz247

Changhai Bian (Barry)_ 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. The ranges for programmable parameters are listed below.

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):

- **Description:** 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.

- **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.

AOOR Mode (Rate Adaptive Atrial Pacing):

- **Description:** This mode provides atrial pacing at a rate that adapts based on the patient's activity level detected by the pacemaker's motion sensor. Pacing is independent of intrinsic atrial 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. The pacing rate shall vary dynamically between the Lower Rate Limit (LRL) and the sensor-defined Maximum Sensor Rate (MSR) based on the patient's detected motion.
    - **Maximum Sensor Rate (MSR):** The Maximum Sensor Rate is the upper limit of the pacing rate that the pacemaker can reach when influenced by the accelerometer's sensor-based activity data. It is independently programmable from the URL
    - **Response Factor:** The parameter controlling how aggressively the pacing rate increases in response to detected activity.
    - **Activity Threshold:** This is the minimum output from the pacemaker's accelerometer sensor that must be exceeded before activity-driven rate modulation begins.
    - **Reaction Time:** The time required for the pacing rate to increase from the Lower Rate Limit (LRL) to the Maximum Sensor Rate (MSR) after the activity threshold is surpassed.
    - **Recovery Time:** The time it takes for the pacing rate to decrease from the Maximum Sensor Rate (MSR) back to the Lower Rate Limit (LRL) once activity ceases.

VOOR Mode (Rate Adaptive Ventricle pacing):

- **Description**: This mode paces the ventricle at a rate responsive to patient activity, detected through motion sensors. The pacing occurs irrespective of intrinsic ventricular 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. The pacing rate shall vary dynamically between the Lower Rate Limit (LRL) and the sensor-defined Maximum Sensor Rate (MSR) based on the patient's detected motion.
    - **Maximum Sensor Rate (MSR):** The Maximum Sensor Rate is the upper limit of the pacing rate that the pacemaker can reach when influenced by the accelerometer's sensor-based activity data. It is independently programmable from the URL
    - **Response Factor:** The parameter controlling how aggressively the pacing rate increases in response to detected activity.
    - **Activity Threshold:** This is the minimum output from the pacemaker's accelerometer sensor that must be exceeded before activity-driven rate modulation begins.
    - **Reaction Time:** The time required for the pacing rate to increase from the Lower Rate Limit (LRL) to the Maximum Sensor Rate (MSR) after the activity threshold is surpassed.
    - **Recovery Time:** The time it takes for the pacing rate to decrease from the Maximum Sensor Rate (MSR) back to the Lower Rate Limit (LRL) once activity ceases.

AAIR Mode (Rate Adaptive and Demand based Atrium Sensing):

2. **Description**: The pacemaker paces the atrium only when no intrinsic atrial activity is detected within a specified interval. It adjusts the pacing rate dynamically based on physical activity detected by a motion sensor, ensuring the atrial pacing rate matches the metabolic demands during exercise or rest.
3. **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.
    - **Maximum Sensor Rate (MSR):** The Maximum Sensor Rate is the upper limit of the pacing rate that the pacemaker can reach when influenced by the

accelerometer's sensor-based activity data. It is independently programmable from the URL

- **Response Factor:** The parameter controlling how aggressively the pacing rate increases in response to detected activity.
- **Activity Threshold:** This is the minimum output from the pacemaker's accelerometer sensor that must be exceeded before activity-driven rate modulation begins.
- **Reaction Time:** The time required for the pacing rate to increase from the Lower Rate Limit (LRL) to the Maximum Sensor Rate (MSR) after the activity threshold is surpassed.
- **Recovery Time:** The time it takes for the pacing rate to decrease from the Maximum Sensor Rate (MSR) back to the Lower Rate Limit (LRL) once activity ceases.

VVIR Mode (Rate Adaptive and Demand based Ventricle Sensing)

4. **Description**: The pacemaker paces the ventricle only when intrinsic ventricular activity is absent. It uses a motion sensor to adapt the pacing rate to the patient's activity level, ensuring appropriate ventricular pacing during physical exertion and rest while inhibiting pacing if natural ventricular beats are detected.
5. **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.
    - **Maximum Sensor Rate (MSR):** The Maximum Sensor Rate is the upper limit of the pacing rate that the pacemaker can reach when influenced by the accelerometer's sensor-based activity data. It is independently programmable from the URL
    - **Response Factor:** The parameter controlling how aggressively the pacing rate increases in response to detected activity.
    - **Activity Threshold:** This is the minimum output from the pacemaker's accelerometer sensor that must be exceeded before activity-driven rate modulation begins.
    - **Reaction Time:** The time required for the pacing rate to increase from the Lower Rate Limit (LRL) to the Maximum Sensor Rate (MSR) after the activity threshold is surpassed.
    - **Recovery Time:** The time it takes for the pacing rate to decrease from the Maximum Sensor Rate (MSR) back to the Lower Rate Limit (LRL) once activity ceases.

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 seven main modules: **Serial Communication Module**, **Parameter Input Module**, **MainStateFlow Module**, **Rate Adaptive Pacing Module**, **Hardware Input Module**, **Hardware Output Module** and **Send Parameters Module**. These design decisions were made to ensure modularity, flexibility, and ease of maintenance while providing clear separation of functionality and responsibilities.

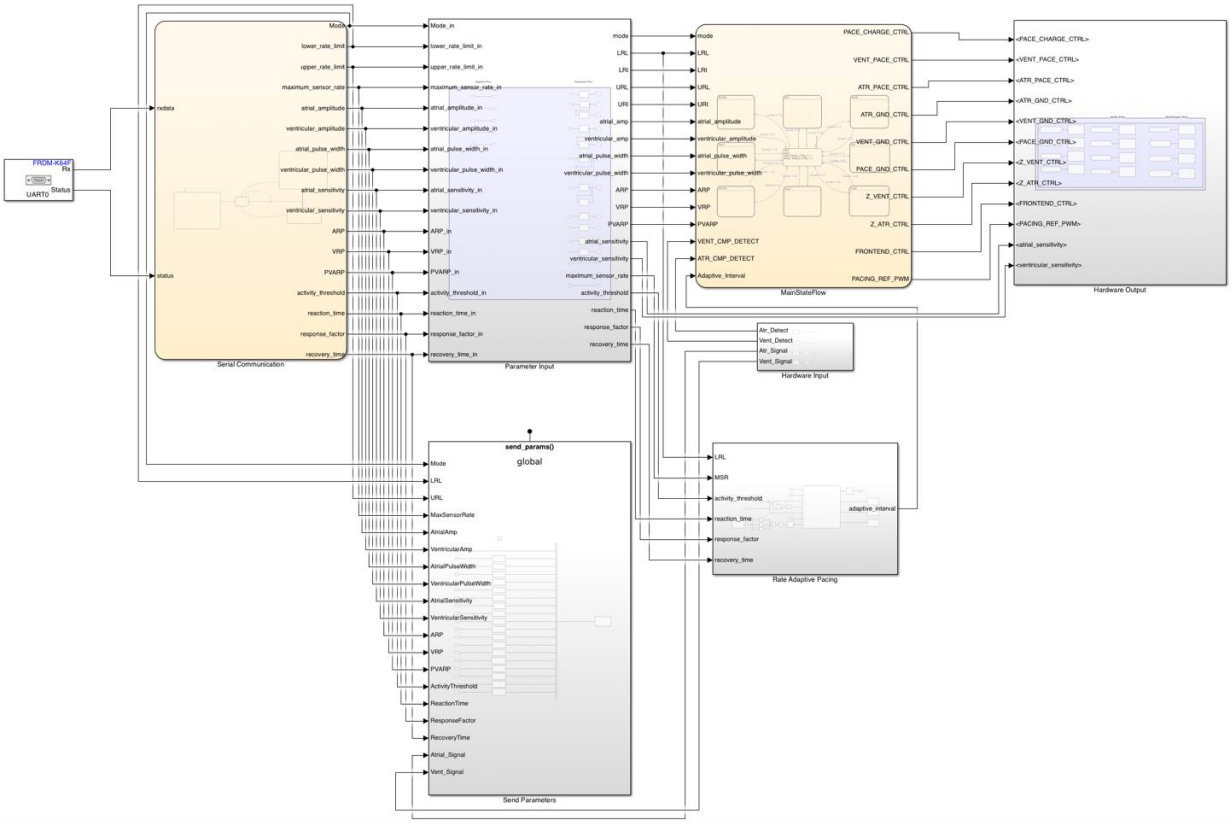


Figure 1. Simulink Block Diagram

Serial Communication Subsystem

This subsystem handles serial communication between the DCM and the Pacemaker. It can process the received data from DCM to Pacemaker, manage default settings when the DCM is disconnected and echo the parameter data to DCM to generate an ECG diagram when the DCM is connected. The processed data is forwarded to both **Send Parameters Module** and **Parameter Input Module**, which subsequently pass it to **MainStateFlow Module**.

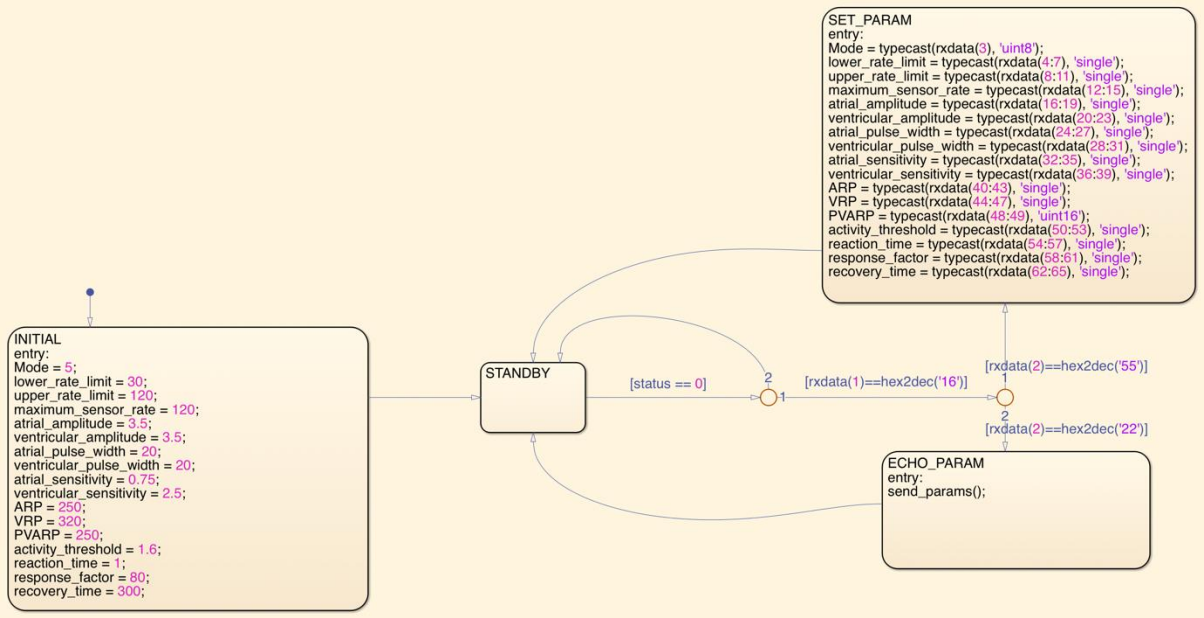


Figure 2. Serial Communication Subsystem

To ensure efficient serial communication, data types for variables were carefully assigned to optimize resource usage. Most variables were allocated as single because these variables requiring decimal precision during calculations in **MainStateFlow Module**. The variable like Mode which wouldn't join in later calculations was allocated as uint8, but if a variable's range exceeded 255, it was assigned uint16. Maintaining the order of transmitted data is also crucial for proper interpretation, and a complete list of all variables in the received sequence is provided below.

Parameter Name	**Data Type**
Mode	uint8(1 byte) -> 0: OFF; 1: AOO; 2: VOO; 3: AAI; 4: VVI; 5: AOOR; 6: VOOR; 7: AAIR; 8: VVIR
lower_rate_limit	single(4 bytes)
upper_rate_limit	single(4 bytes)
maximum_sensor_rate	single(4 bytes)
atrial_amplitude	single(4 bytes)
ventricular_amplitude	single(4 bytes)
atrial_pulse_width	single(4 bytes)
ventricular_pulse_width	single(4 bytes)
atrial_sensitivity	single(4 bytes)
ventricular_sensitivity	single(4 bytes)
ARP	single(4 bytes)
VRP	single(4 bytes)
PVARP	uint16(2 bytes)
activity_threshold	single(4 bytes)
reaction_time	single(4 bytes)

response_factor	single(4 bytes)
recovery_time	single(4 bytes)

Table 1. Received Serial Communication Variables

Parameter 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

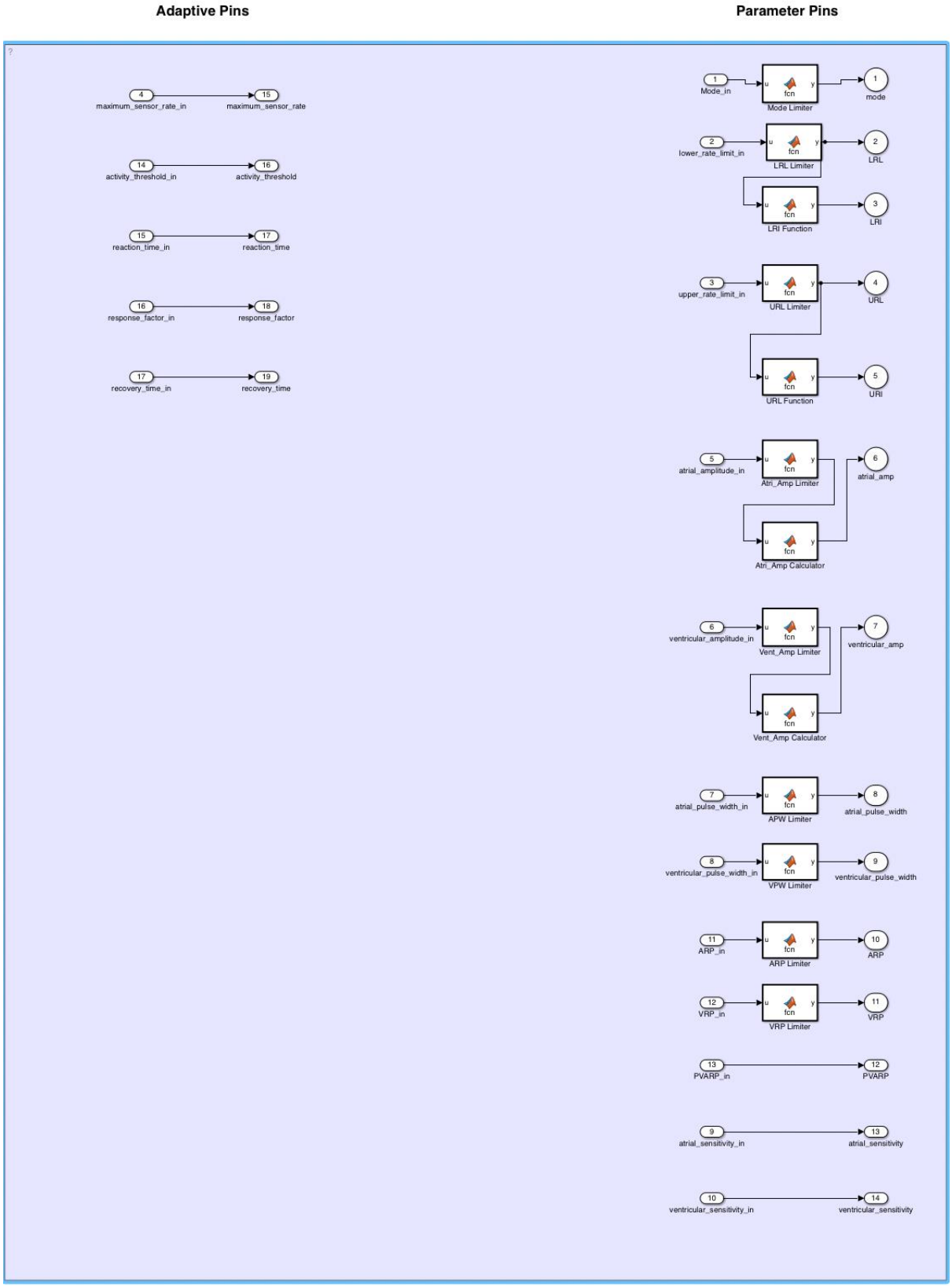


Figure 3 ensures that data is correctly processed before it reaches the decision-making module.

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 **0-8.**
    - **Lower Rate Limit (LRL)**: Ranges between **30-175 BPM**.
    - **Upper Rate Limit (URL)**: Ranges between **50-175 BPM**.
    - **Atrial/Ventricular Amplitude**: Ranges between **0-5 V**.
    - **Atrial/Ventricular Pulse Width**: Ranges between **0.1-1.9 ms**.
    - **Atrial Refractory Period (ARP)**: Ranges between **150-500 ms**.
    - **Ventricular Refractory Period (VRP)**: Ranges between **150-500 ms**.
    - **Atrial/Ventricular Sensitivity**: Ranges between **1.0-10 mV**.
    - **Maximum Sensor Rate**: Ranges between **50-175 ppm**.
    - **Activity Threshold**: Ranges between **50-175 ppm**.
    - **Reaction Time**: Ranges between **10-50 sec**.
    - **Response Factor**: Ranges between **1-16**.
    - **Recovery Time**: 2-16 **min**.

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.

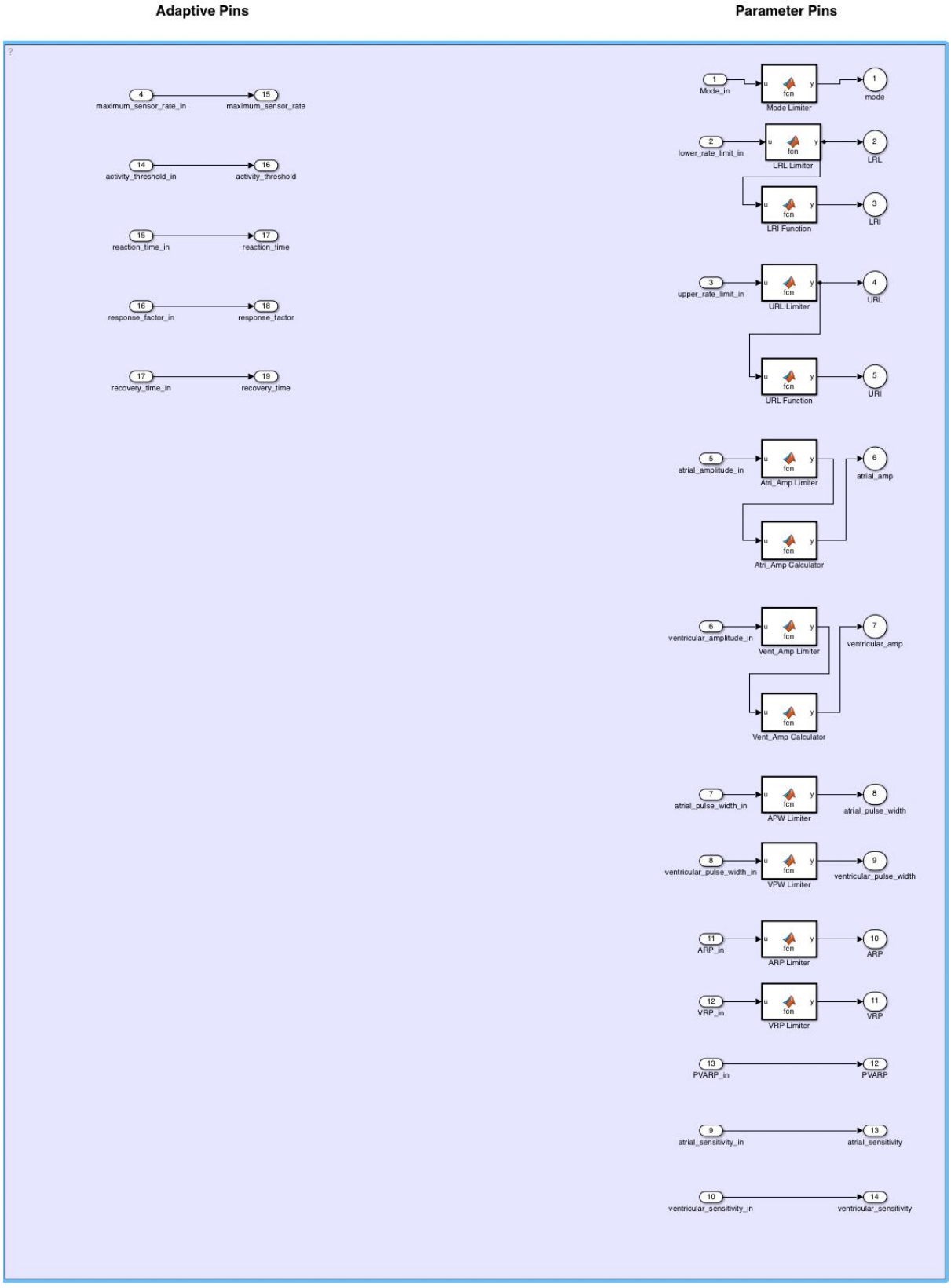


Figure 3. Parameter Input Diagram

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

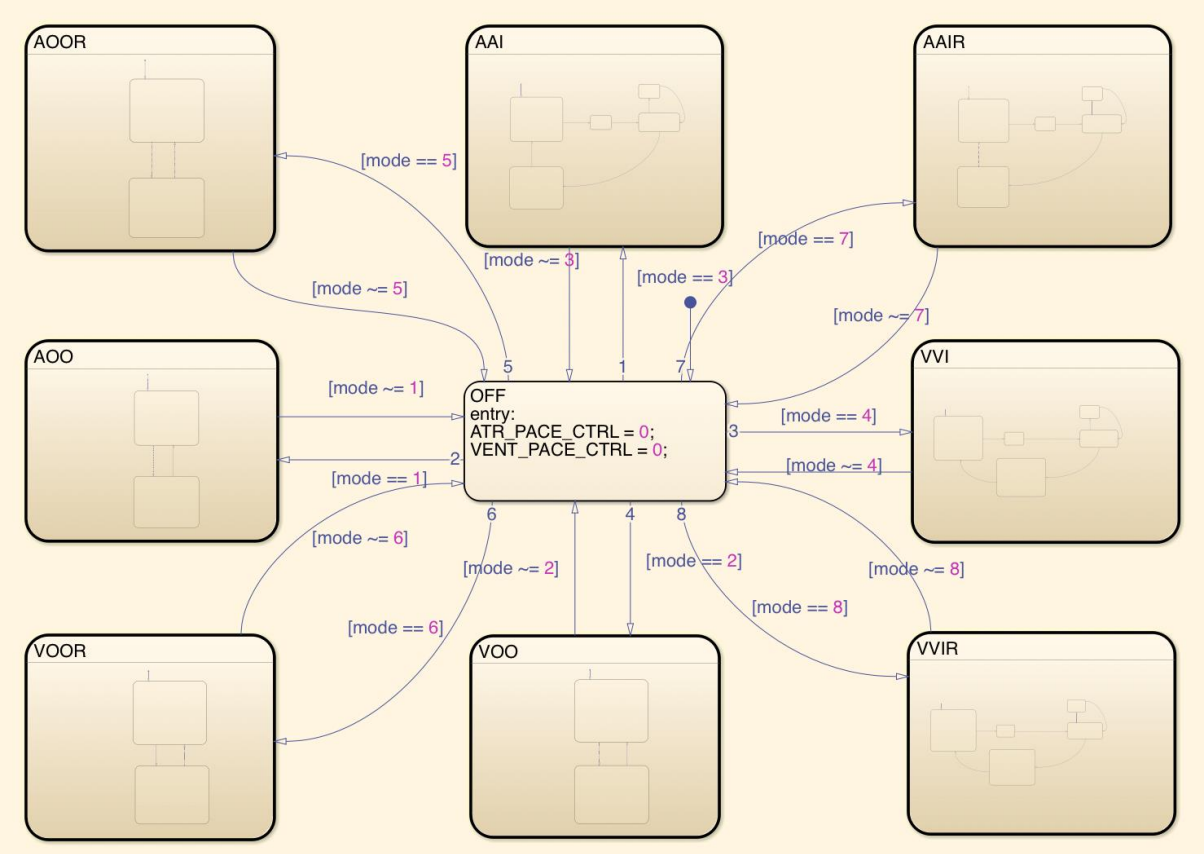


Figure 4. It determines which pacing mode to activate (AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, VVIR) and controls pacing actions, such as triggering pacing pulses.

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.

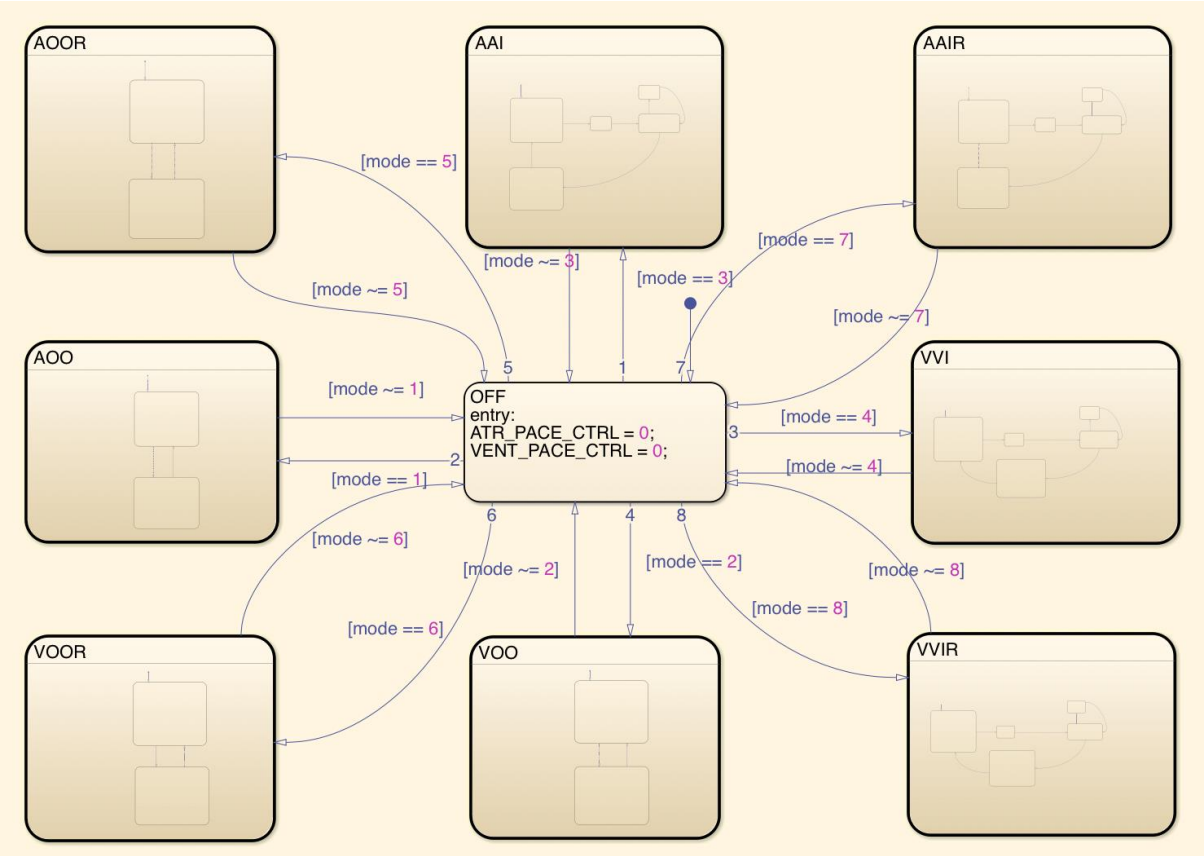


Figure 4. Stateflow Chart for Pacemaker Modes

Rate Adaptive Pacing Subsystem:

This subsystem monitors activity using the onboard accelerometer and adjusts the pacing rate based on the detected activity level. As shown in Figure 5 below, preprocessing is performed to calculate the sensor-indicated rate, the rate of increase, the rate of decrease, and to determine if the detected activity surpasses the activity threshold.

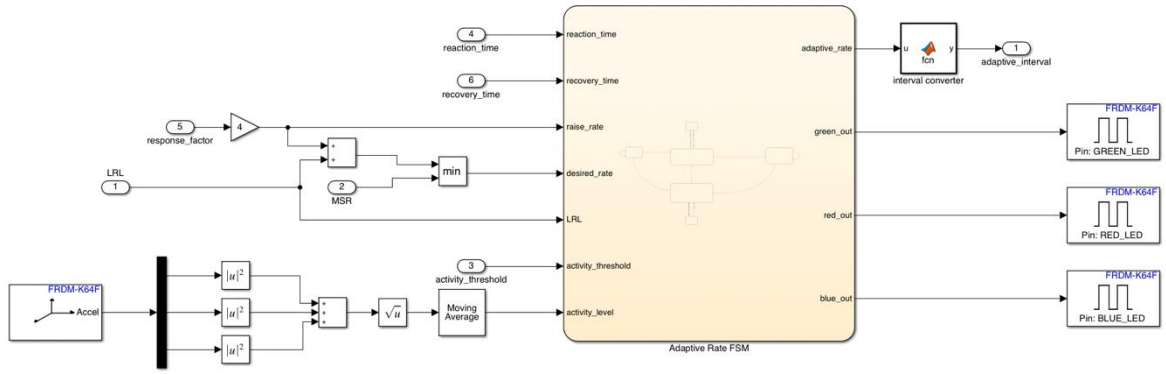


Figure 5: Preprocessing inputs for Rate Adaptive stateflow

Determining Activity Level:

To assess the activity level, the magnitude of the accelerometer's raw data is calculated and stored in a buffer. This approach ensures that each data point from the accelerometer is processed while allowing comparisons with previous data points.

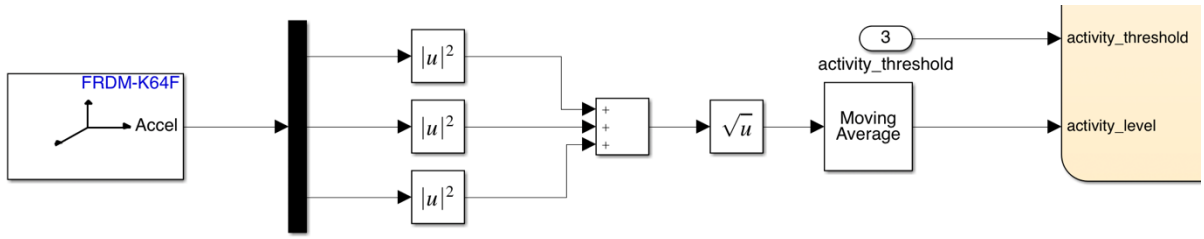


Figure 6: Activity Level Preprocessing

Rate Adaptive Stateflow:

The rate-adaptive logic evaluates whether the activity level surpasses the activity threshold or remains below it. Based on this evaluation, it transitions to the appropriate state to either increase or decrease the pacing rate linearly over time until the sensor-indicated rate is reached or the rate returns to the LRL.

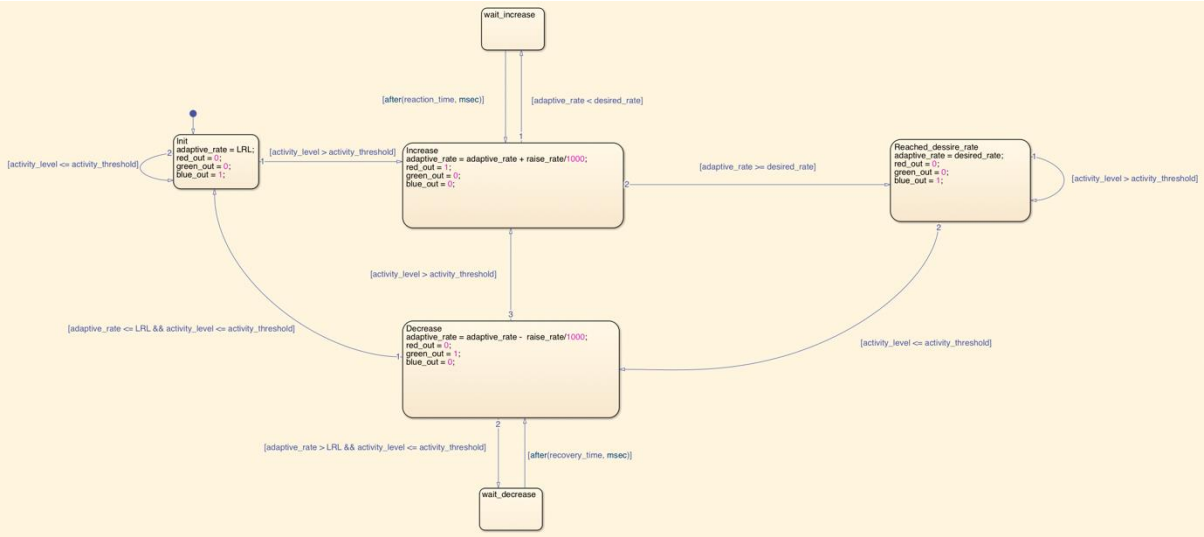


Figure 7: Rate Adaptive Logic

Parameter 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.

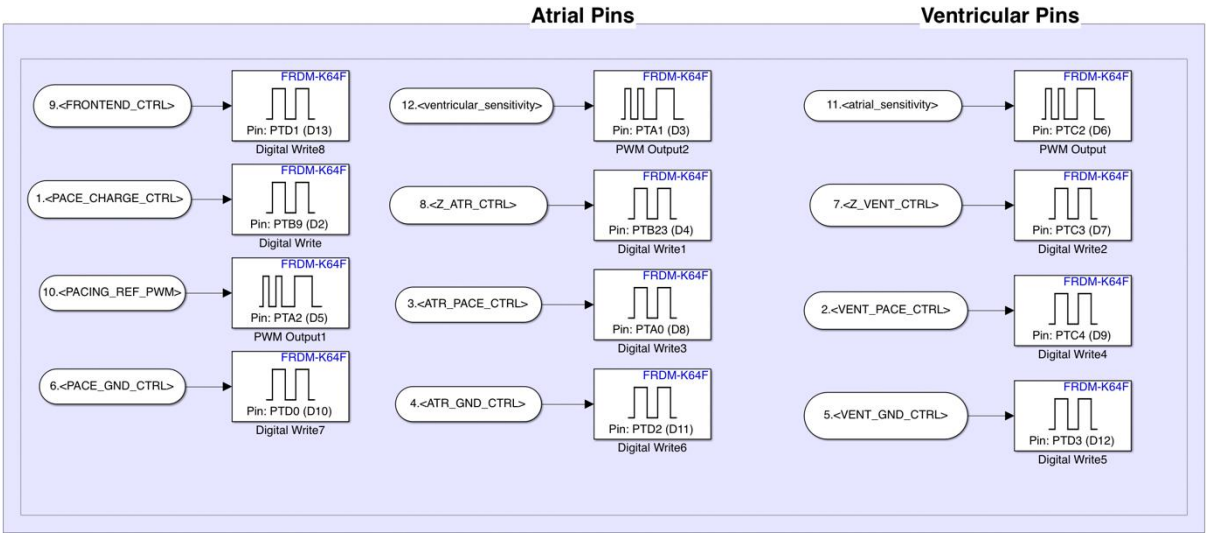
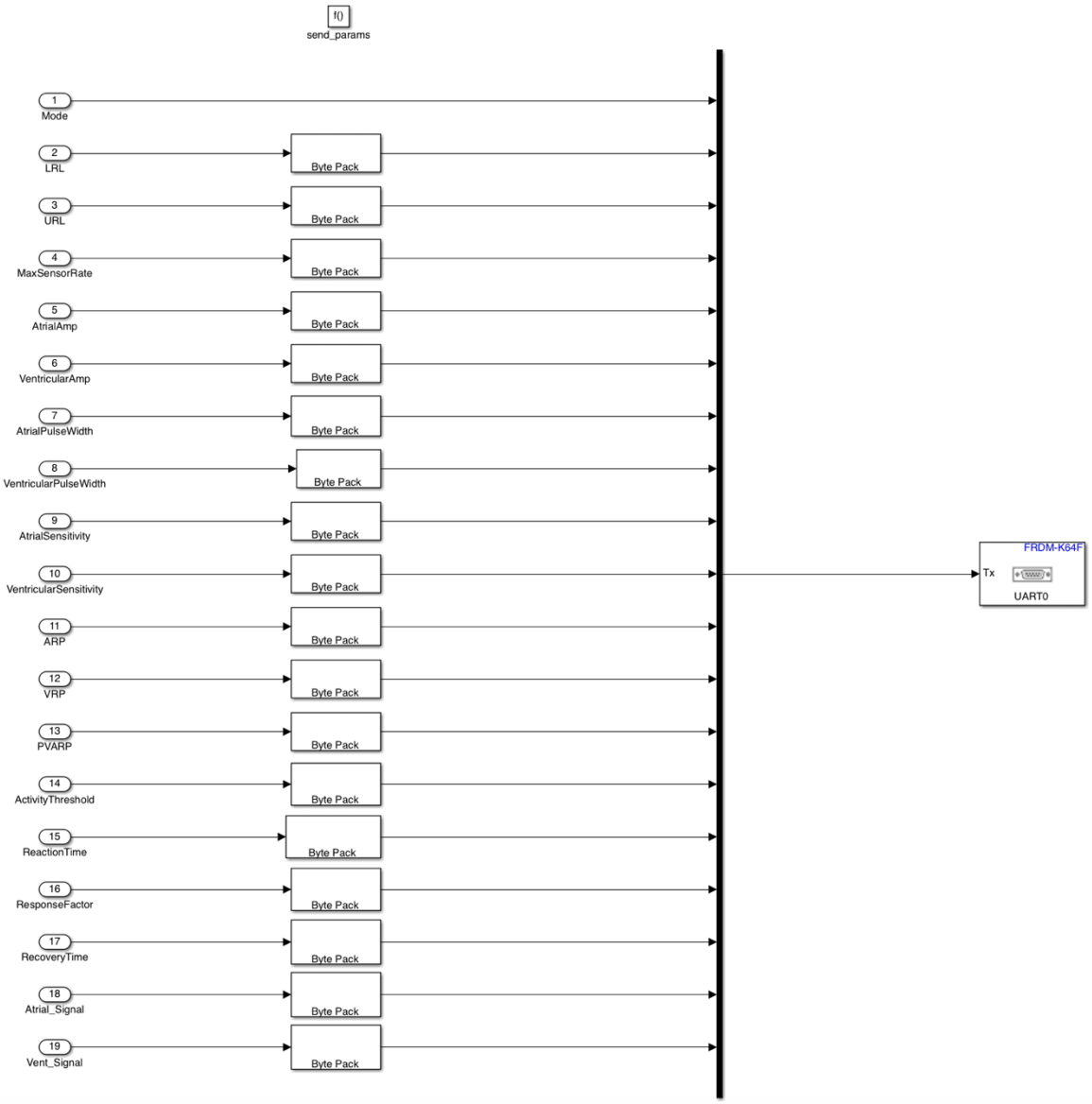


Figure 8. Detailed Pin Assignment Diagram

Send Parameters Module:

The send parameter module is a subsystem designed to transmit system parameters and signals through serial communication. It is triggered by the ECHO_PARAM state in the serial communication module when the condition [rxdata(2) == hex2dec('22')] is met. The module takes various input parameters (e.g., Mode, LRL, URL, etc.), processes them through Byte Pack blocks, and sends the formatted data via UART. Notably, the Atrial_Signal and Vent_Signal are included to generate ECG waveforms, making them essential for monitoring. This subsystem ensures efficient and accurate data exchange with the DCM.



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.

Rate-Adaptive Modes: AOOR and VOOR: AOOR (Atrial Rate-Adaptive Pacing): This mode provides continuous atrial pacing while dynamically adjusting the pacing rate based on the patient's activity level detected via motion sensors. It is particularly beneficial for patients with chronotropic incompetence, ensuring that their heart rate adapts to varying metabolic demands. VOOR (Ventricular Rate-Adaptive Pacing): Similar to AOOR, this mode ensures continuous ventricular pacing with a rate modulated by the patient's activity. It is useful in situations where reliable ventricular support is critical, and metabolic needs vary throughout the day.

Hybrid Modes: AAIR and VVIR: AAIR (Rate-Adaptive Atrial Sensing with Inhibition): This mode senses intrinsic atrial activity and paces the atrium only when no natural activity is detected. The pacing rate adapts dynamically to physical activity, optimizing cardiac output while conserving energy when natural activity is sufficient. VVIR (Rate-Adaptive Ventricular Sensing with Inhibition): Similarly, this mode senses ventricular activity and delivers pacing only in its absence. It adjusts pacing rates based on the patient's detected activity levels, ensuring optimal support during exertion while avoiding unnecessary pacing during rest.

Flexibility: By implementing these modes within a MainStateFlow, each can be adjusted or updated independently without affecting others. This modular design allows for smooth transitions between modes, ensuring that the pacemaker can adapt to changing patient needs. For example, a patient's condition might necessitate switching from AAIR to VVIR or AOOR depending on changes in intrinsic activity or metabolic demands.

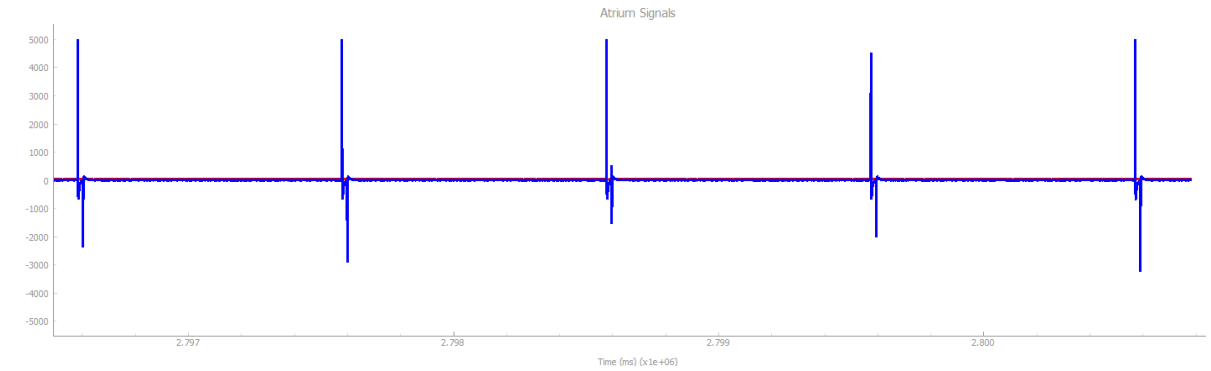
Safety and Efficiency: The clear distinction between asynchronous, demand-based, and rate-adaptive modes ensures that each type of pacing is optimized for its intended purpose. This not only maximizes patient safety by providing reliable pacing in critical situations but also improves battery efficiency, extending the device's operational lifespan. By aligning pacing strategies with physiological needs, the system reduces the likelihood of adverse events such as unnecessary pacing or pacemaker syndrome.

3. Validation and Verification

Validation was completed on each of the 8 possible pacing modes to make sure that they behaved as expected.

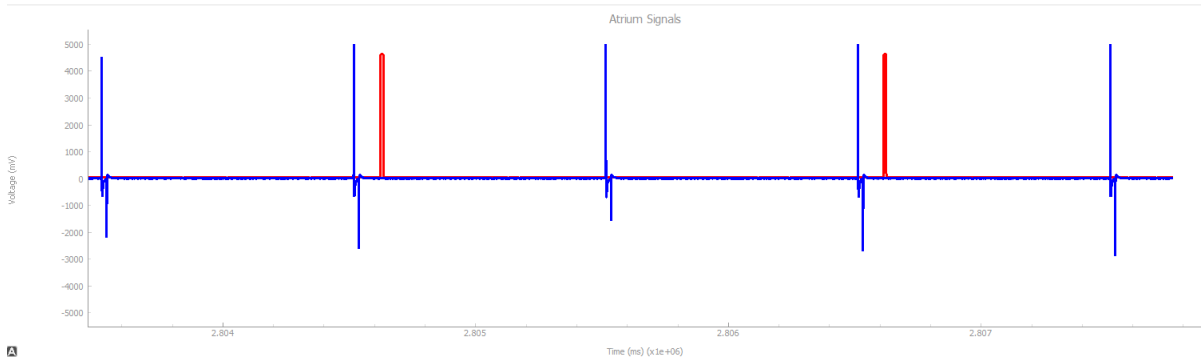
AOO

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. The pacemaker should pace every second. This test was passed successfully.

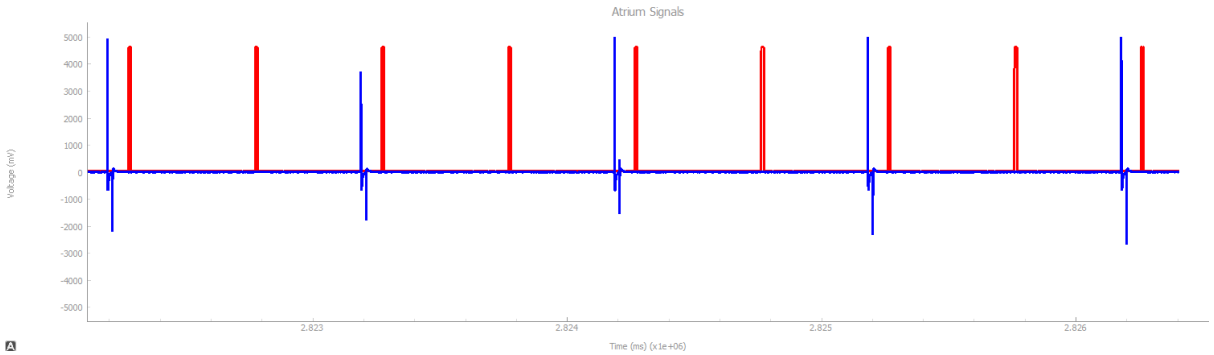


9 Graph of the atrium showing artificial pacing at 60 BPM

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. It should pace at 60 BPM no matter what the natural heart is doing. Both tests passed as shown below:



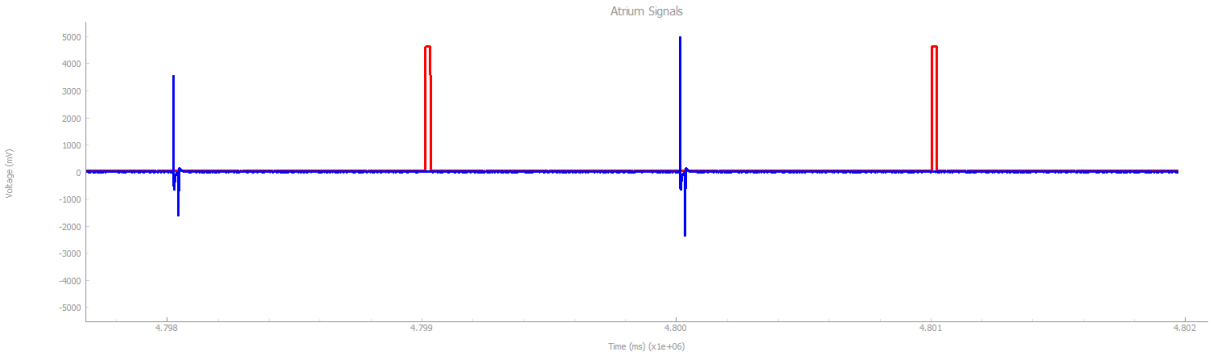
10 Graph of atrium with natural heart rate set to 30 BPM



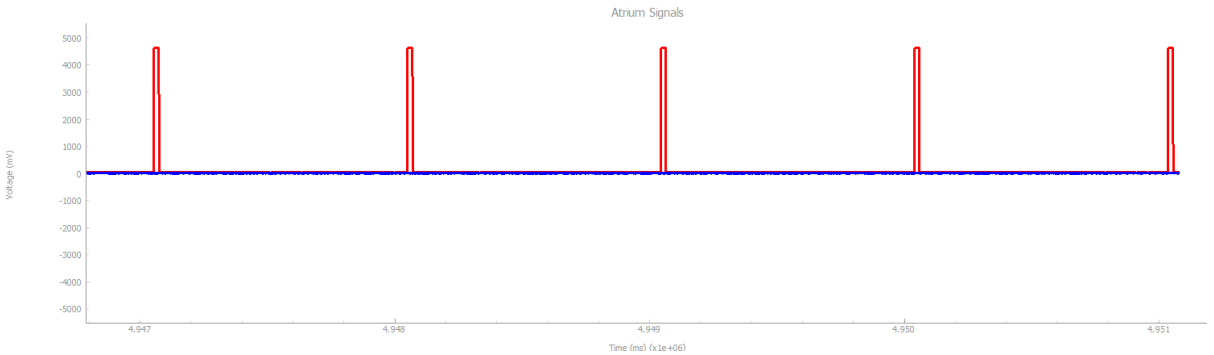
11 Graph of atrium with natural heart rate set to 120 BPM

AAI

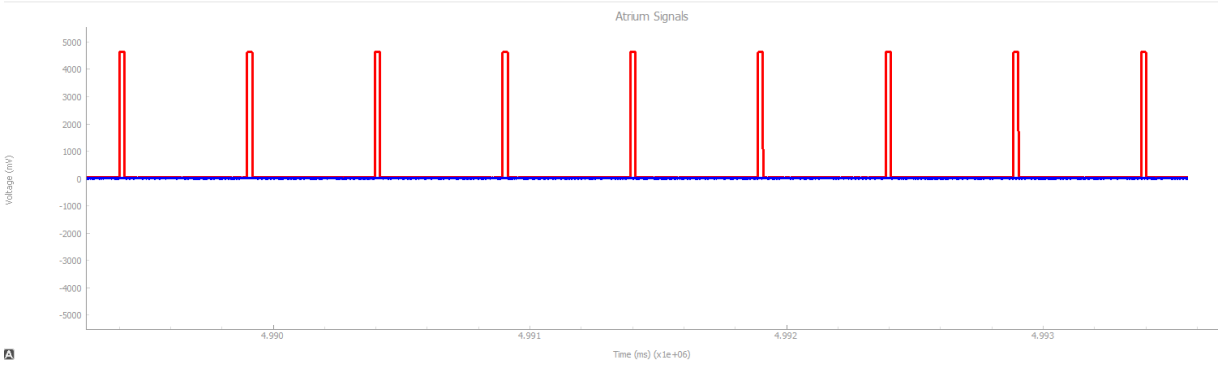
To test the sensing capabilities of this mode, the target heart rate was set to 60 BPM and natural heart rate was first set to 30 BPM, which is much lower than the target. It is expected that pacing should occur. It was then set to 60 BPM, which is the point where pacing should stop since the target 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.



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



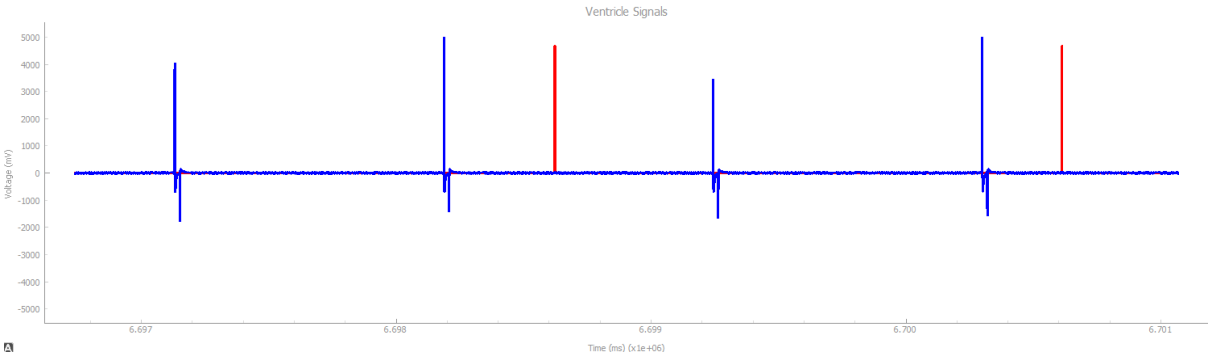
13 Graph of Atrium with natural heart rate set to 60 BPM



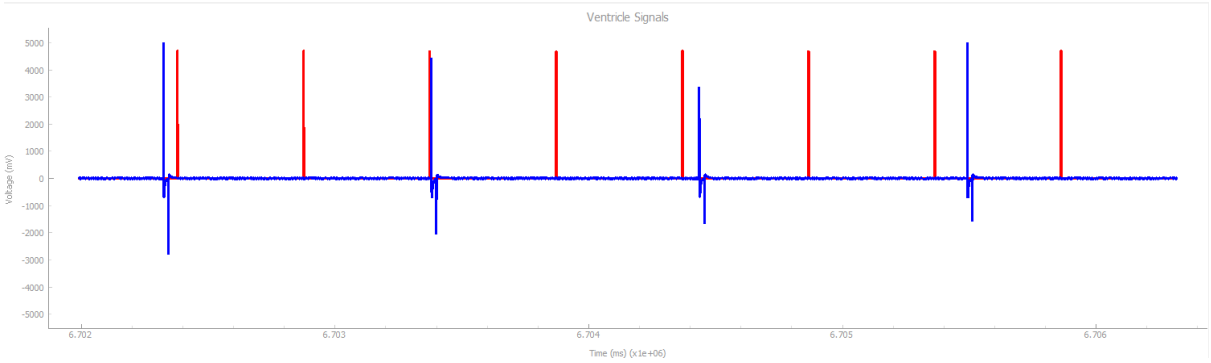
14 Graph of Atrium with natural heart rate set to 120 BPM

VOO

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



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

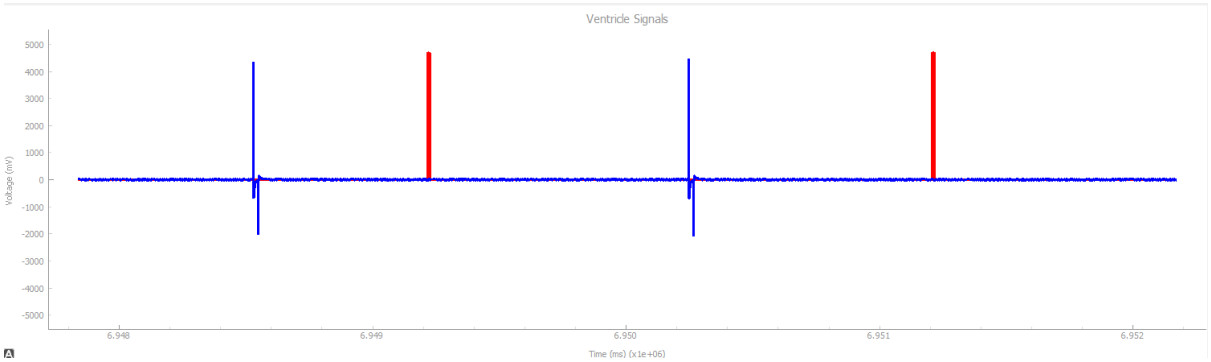


16 Graph of ventricle with natural heart rate set to 60 BPM

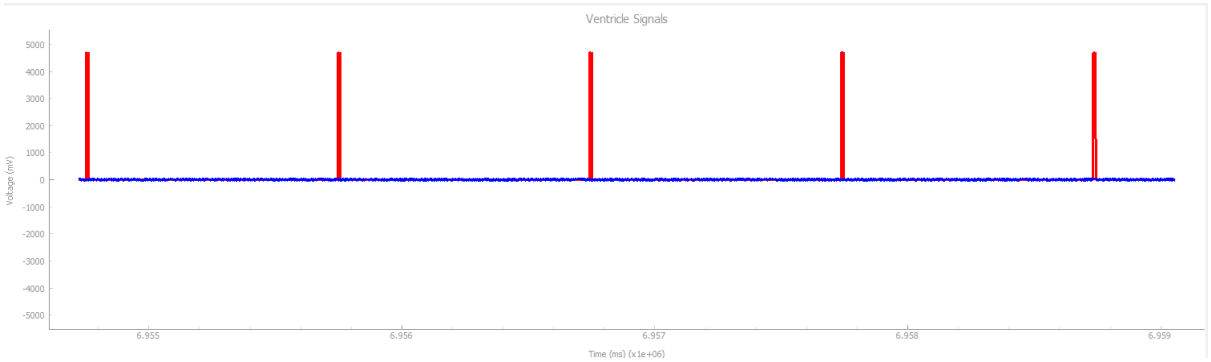
VVI

The testing for this mode is the same as the testing for AAI, since the expected pacing for the ventricle is the same. This mode passed all the tests and was able to sufficiently react to changes

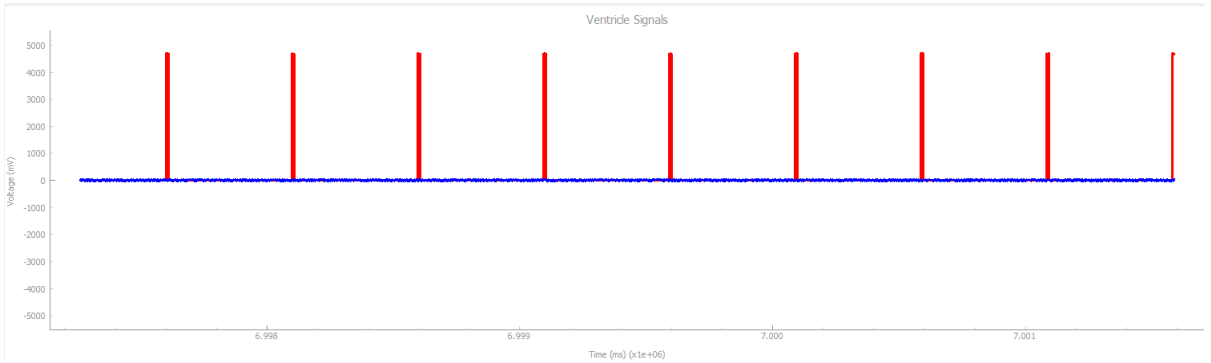
in the natural heart rate as shown in the graphs below:



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



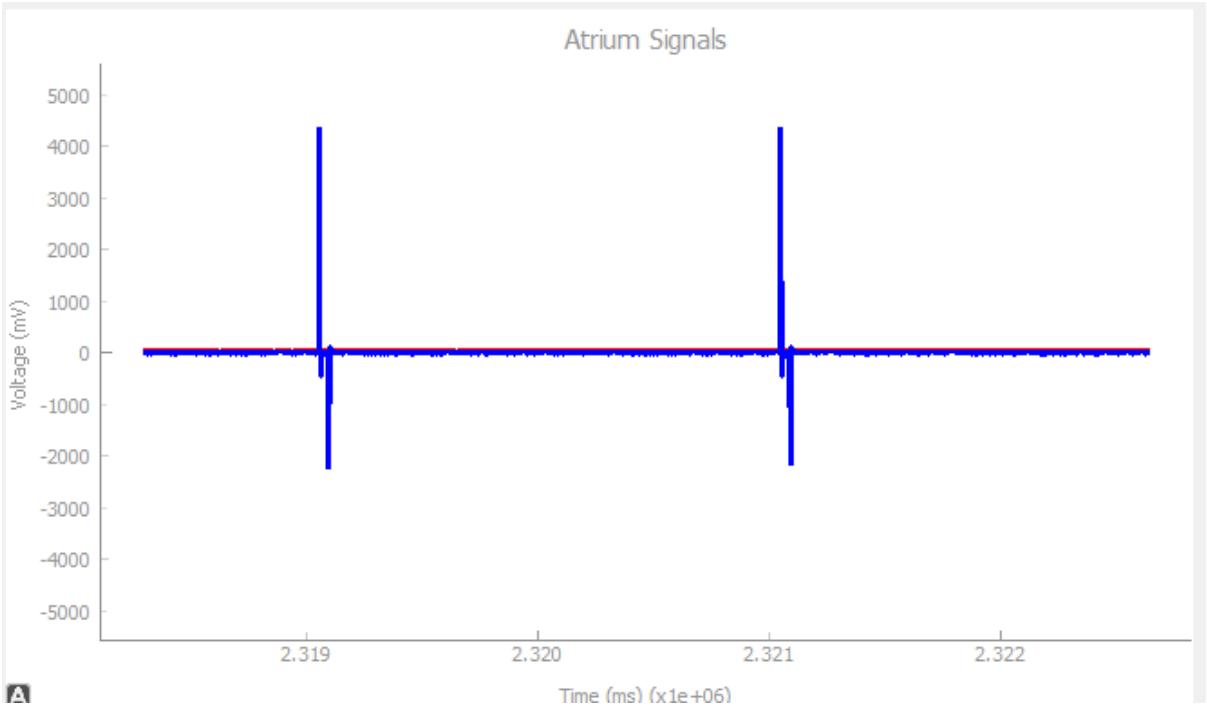
18 Graph of ventricle with natural heart rate set to 60 BPM



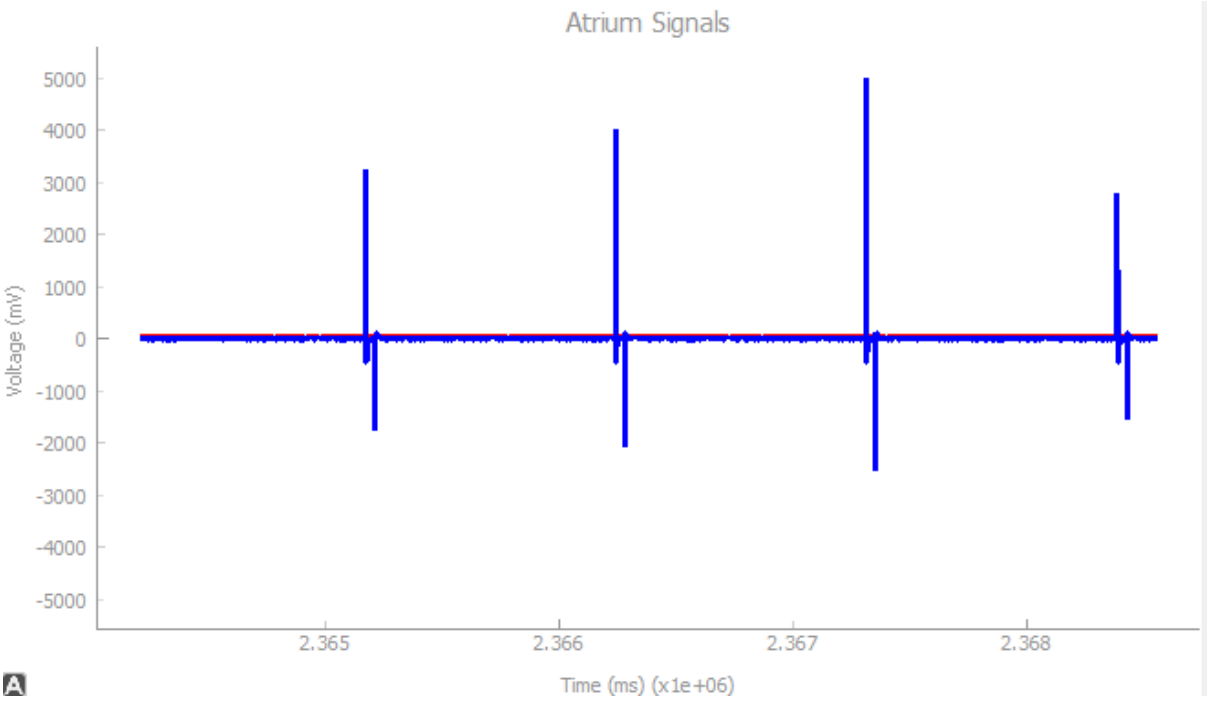
19 Graph of ventricle with natural heart rate set to 120 BPM

AOOR

The first test that needs to be run for this mode is a test of the motion sensing and the resulting changes in heart rate due to motion. To test this, the lower limit for paces per minute (PPM) was set to 30 and the upper limit was 60. This means that the heart rate should double after shaking the pacemaker. This occurred as shown in the graphs of the atrium below.

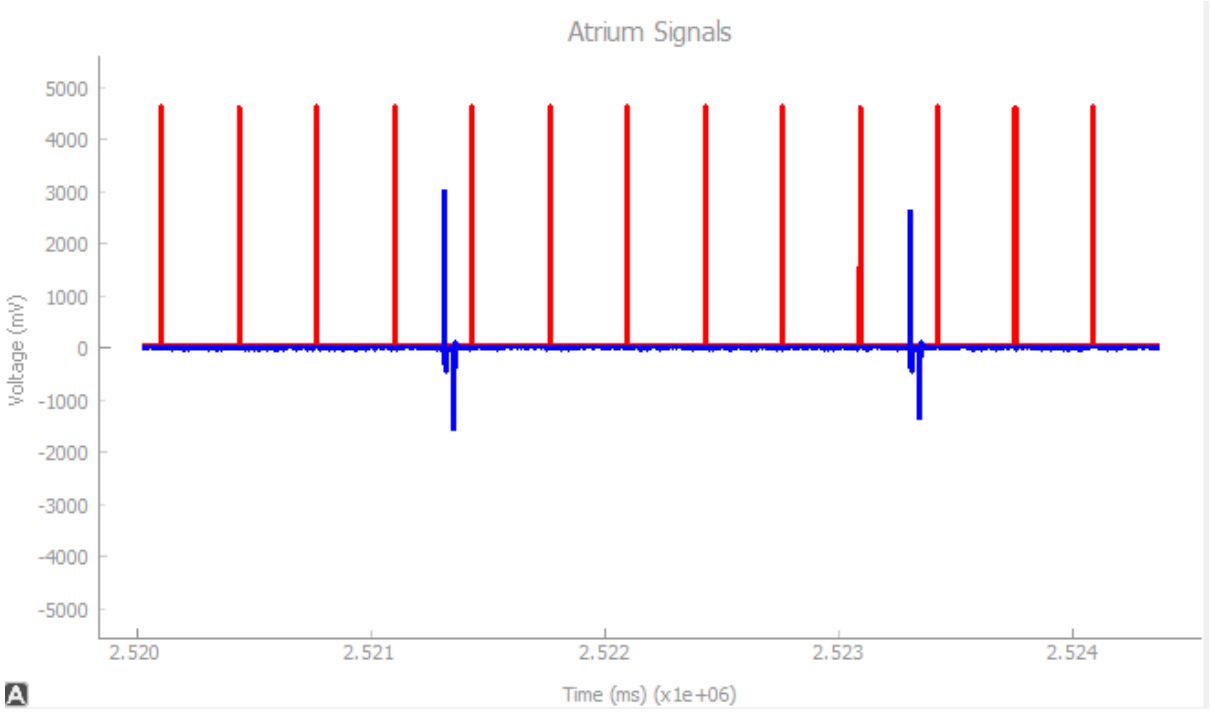


20 Pacing rate before shaking



21 Increased pacing rate after shaking

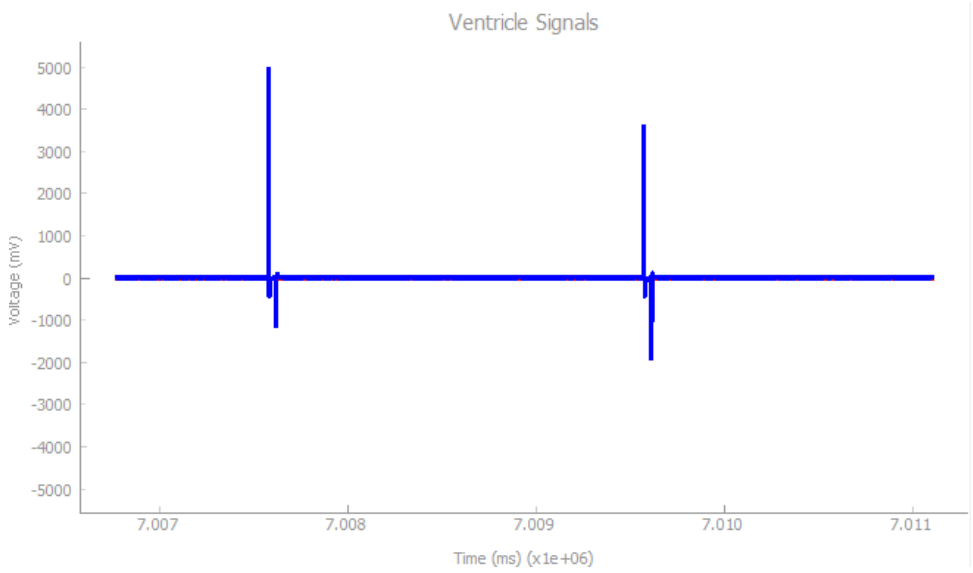
The next test is to make sure that the pacemaker doesn't inhibit pacing due to natural beats from the heart. To test this, the lower PPM was still set to 30, but the simulated heart was set to beat at 180 BPM. It is expected that the pacing does not decrease no matter how high the natural heart rate is and this is shown in the graph below:



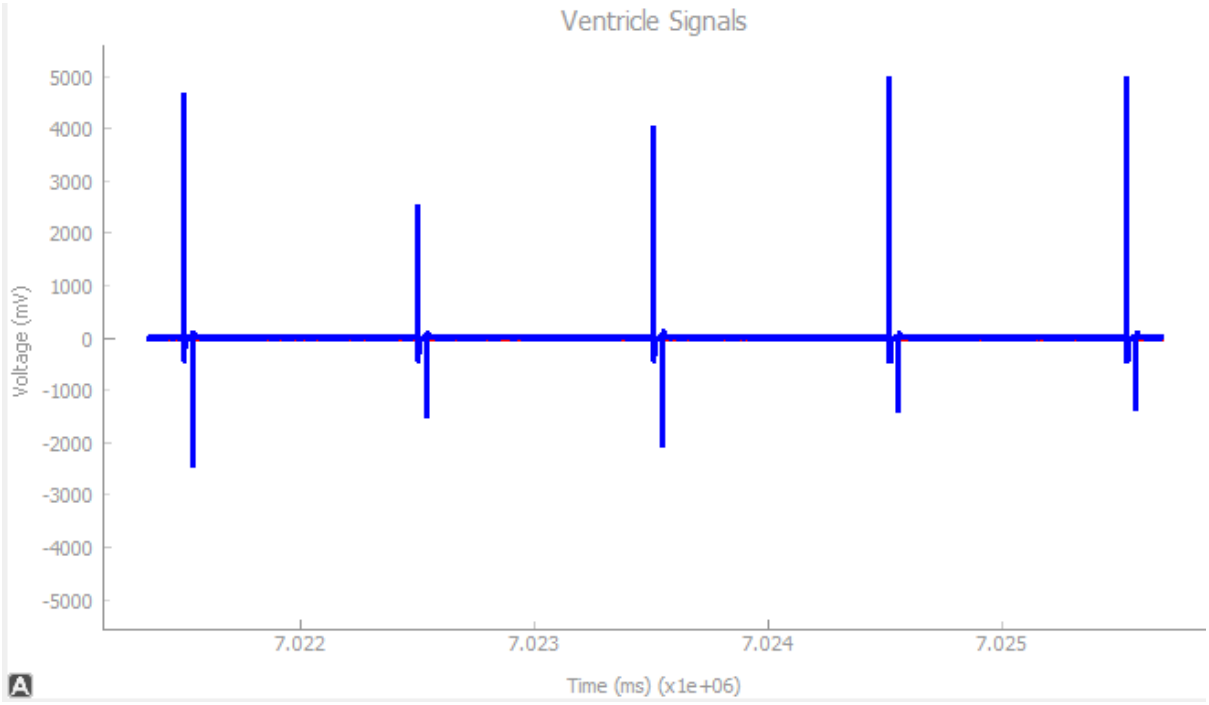
22 Graph of pacing with natural heartbeat

VOOR

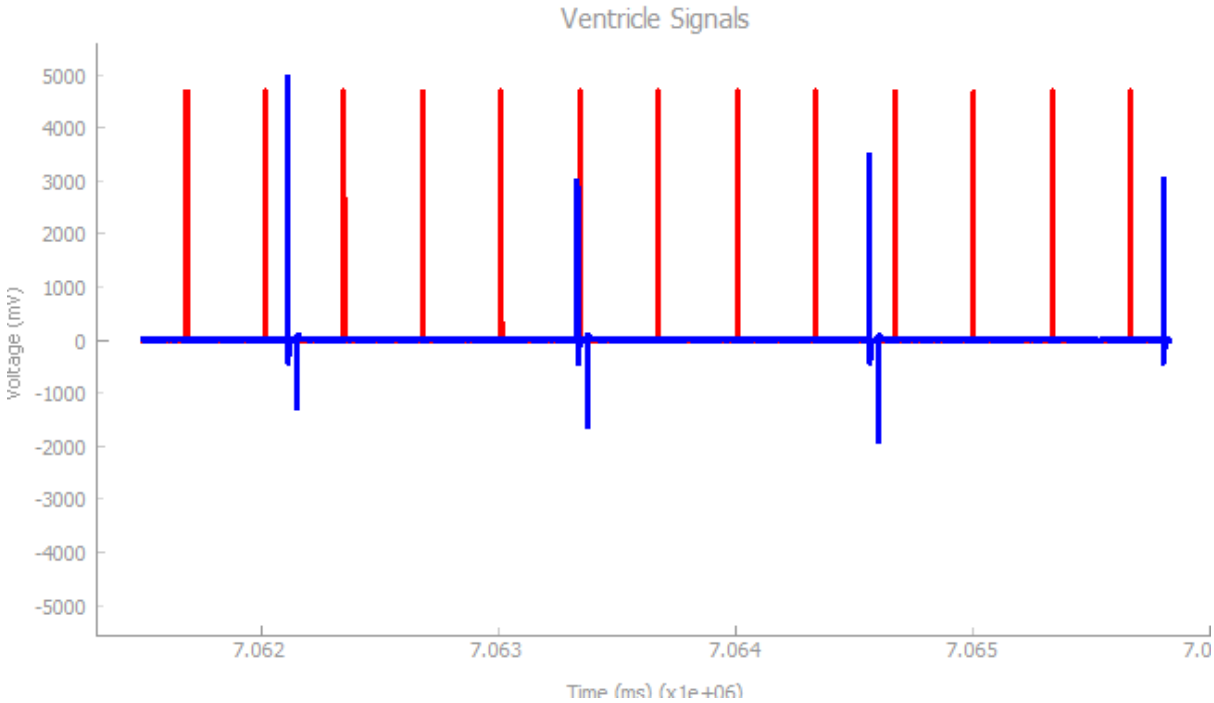
The testing for VOOR is the same as the testing for AOOR above where the PPM should change based on motion and it shouldn't change its pacing behaviour based on the activity of the normal heart. All 3 tests passed as expected, as shown in the graphs below:



23 Pacing with no motion



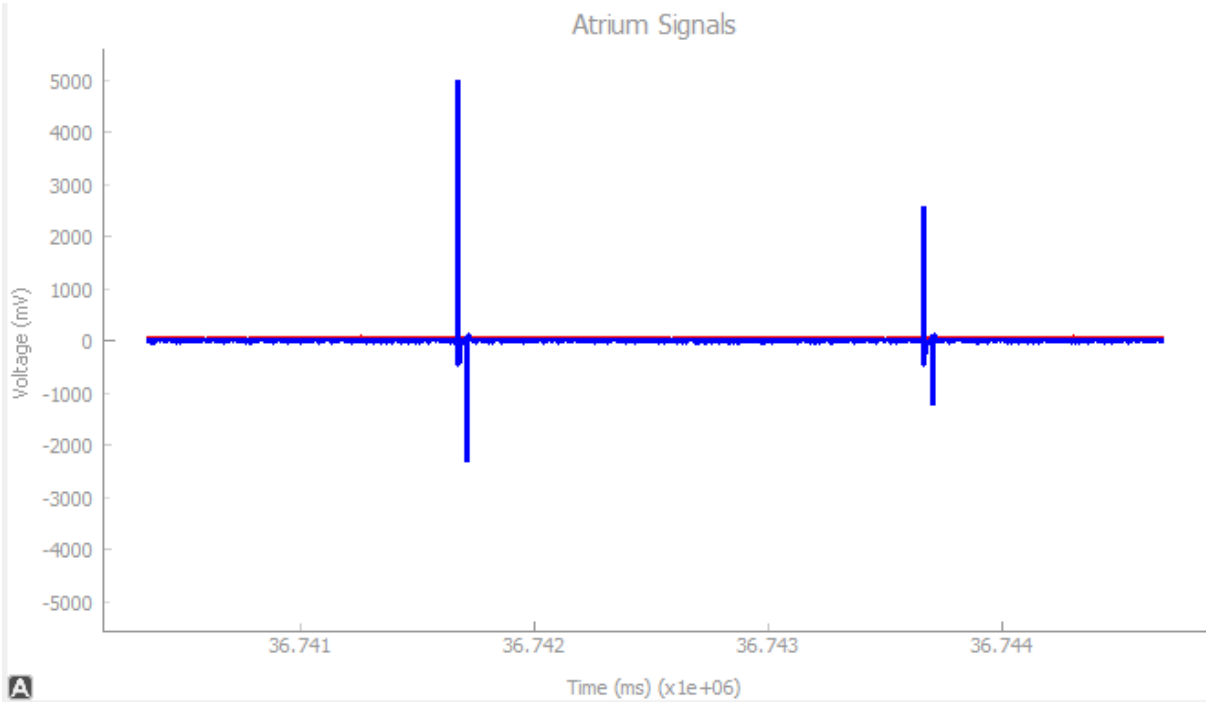
24 Pacing after applying motion



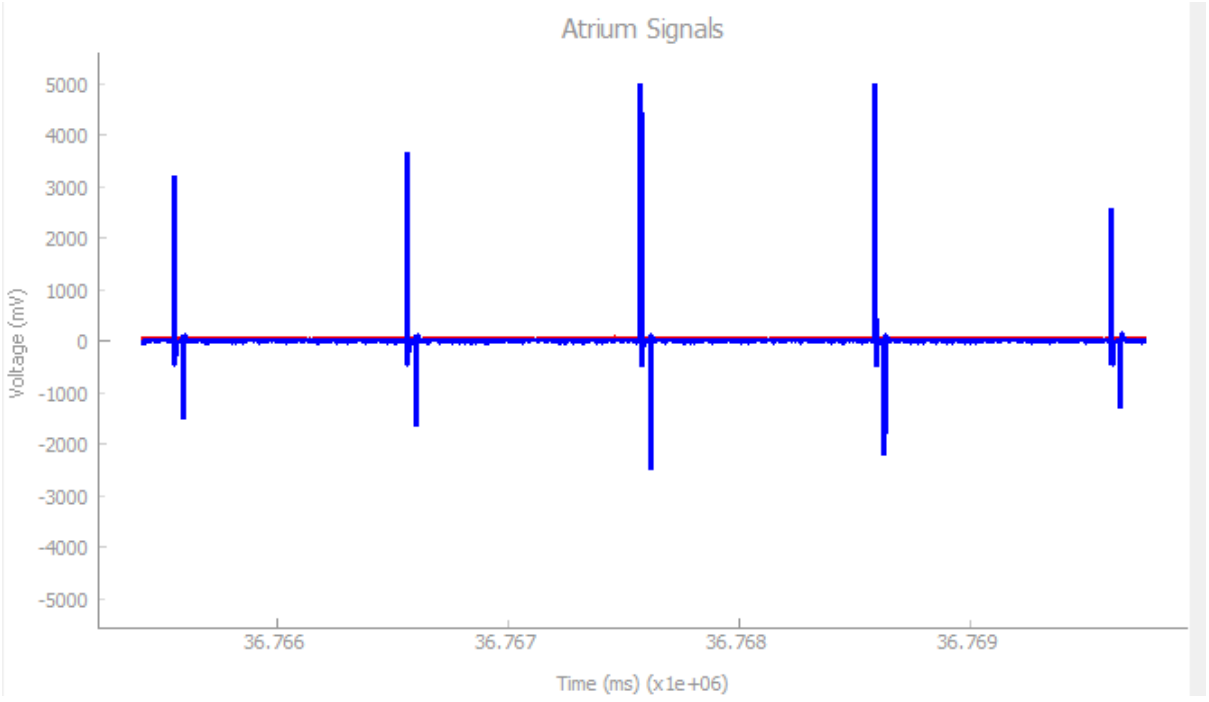
25 Graph of pacing with natural heart rate

AAIR

For AAIR, there are 3 tests that need to be conducted. The first is to check that motion sensing works and adjusts rate sensing accordingly. To test this the LRL will be set to 30 and then pacemaker will be shaken to simulate motion. The heart rate should go up accordingly. This test was a success as shown below:

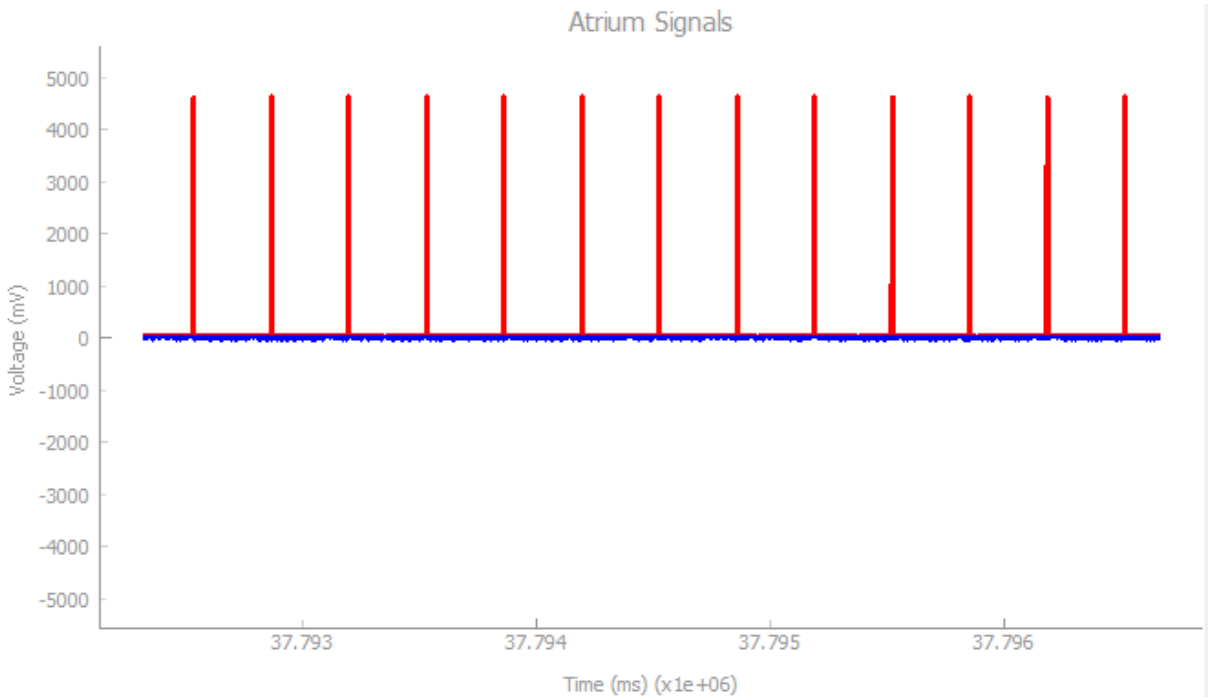


26 Pacing without motion



27 Pacing with motion

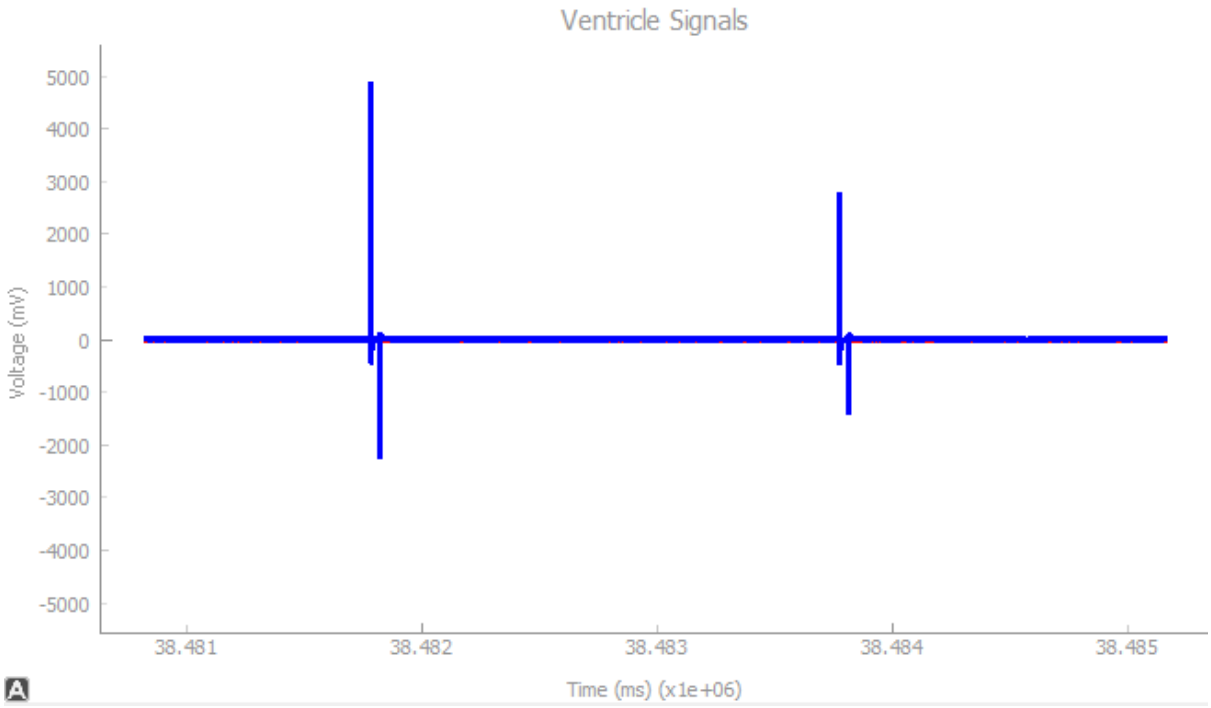
The second test that must be completed focuses on the inhabitation of pacing if the heart is beating fast enough. To test this, the heart was set to 180 BPM and the pacemaker was shaken to simulate movement. The expected result is that the pacemaker doesn't pace since the heartrate is already high enough. It passed this test as shown below:



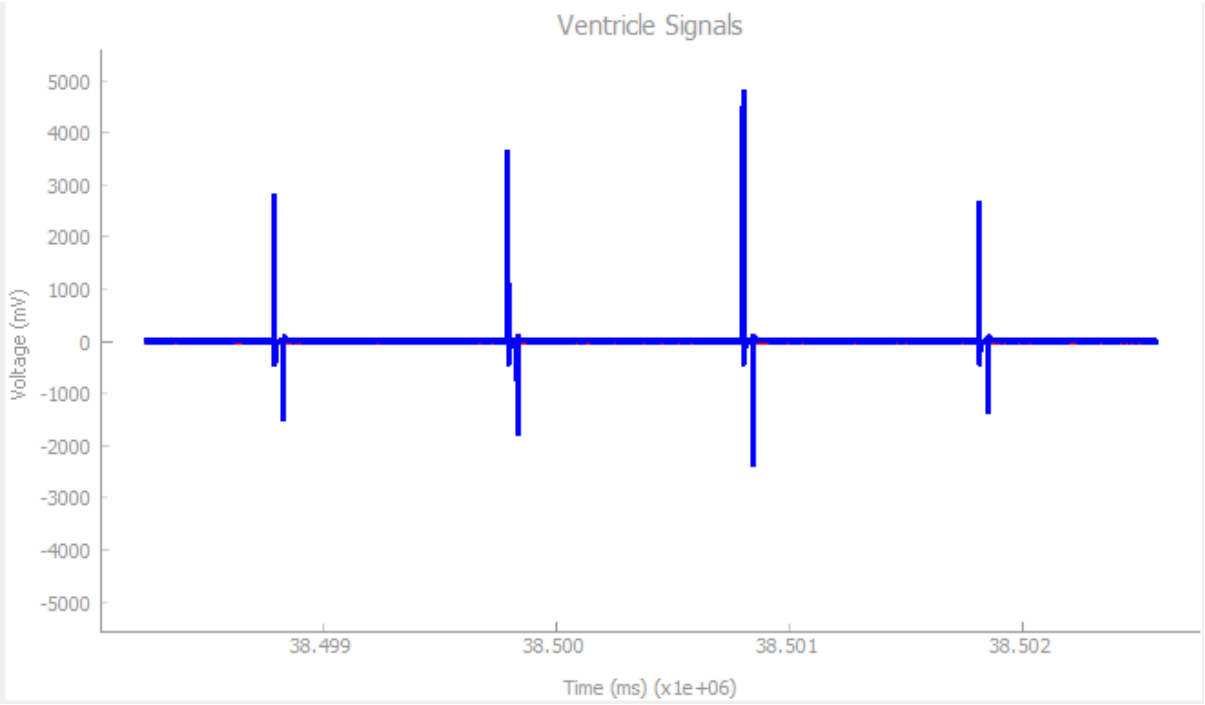
28 Pacing with high BPM and motion

VVIR

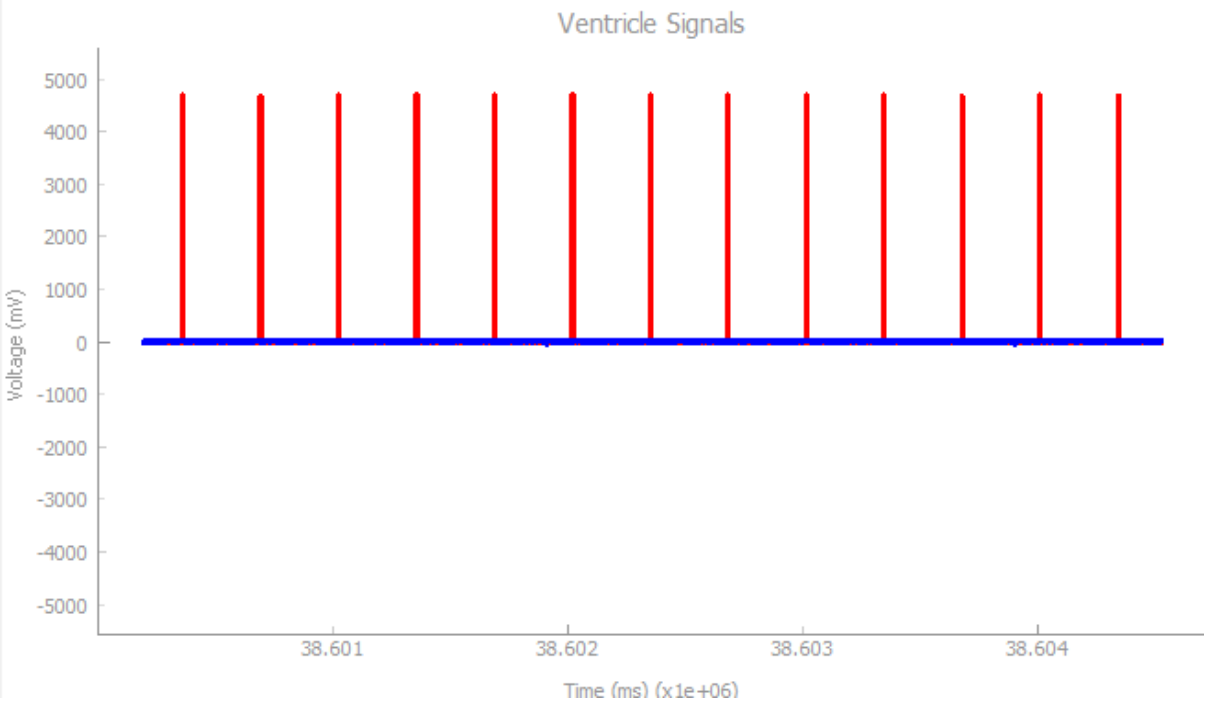
The same tests used to test this mode are the same the ones used to test AAIR. This mode also passed all of the tests as shown below:



29 Pacing without motion



30 Pacing with motion



31 Pacing with high BPM and motion

Input Limiters

The input limiters are connected to all of the inputs that the pacemaker receives to make sure that it is impossible to program a pacemaker in an unsafe way using the DCM. Every single input was tested with both safe and unsafe ranges and the input limiters successfully limited the parameters to safe values.

Assurance

The pacemaker has been thoroughly safety tested by first testing every single mode individually and making sure it acts as expected and doesn't produce unsafe behaviour. Then the parameters are limited in such a way that the pacemaker cannot produce dangerous behaviour when it receives the wrong input. The only way that the system can harm the patient is through gross negligence from staff who are installing the pacemaker or programming it. Since the pacemaker can cause harm if it is set to pace the wrong part of the heart, even if the pacing mode itself is safe since it must be activated on only patients who need it.

4. Requirements Changes

As the pacemaker system evolves, the requirements have been updated to incorporate additional functionality and accommodate future advancements. The primary change is the integration of serial communication between the DCM and the pacemaker, enabling real-time transmission and storage of programmable parameter data, as well as verification of stored values to ensure accuracy. This change introduces a need for robust handling of communication errors and secure data management. Another update involves the inclusion of electrogram (egram) visualization in the DCM, requiring real-time data reception and display capabilities for both atrial and ventricular signals, thus enhancing monitoring functionality.

Furthermore, the rate-adaptive pacing modes now track physical activity using the onboard accelerometer, necessitating the design and implementation of dynamic pacing intervals. These changes required fine-tuning parameters such as sensitivity and MSR to adapt more effectively to patient activity. Finally, programmable parameters, including pulse width, amplitude, and refractory periods, have been extended to accommodate a wider range of values, ensuring the system's flexibility to address diverse patient needs. These changes highlight the evolving complexity of the pacemaker system to ensure it remains adaptable and effective for broader use cases.

In addition, future updates to the pacemaker system will include advanced dual-chamber pacing modes like DDD and DDDR, requiring synchronization of atrial and ventricular pacing with programmable AV delays. Enhanced sensors, such as those measuring respiratory rate or blood oxygen levels, will enable more comprehensive rate-adaptive pacing, demanding seamless integration of new data types. Real-time adjustments to parameters like pulse amplitude and refractory periods will improve responsiveness to patient needs, while cloud-based telemetry will facilitate remote monitoring, requiring secure communication protocols to ensure data privacy.

5. Design Decisions Changes

The MainStateFlow Module was significantly updated to incorporate all required pacing modes, including AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR. This involved integrating inputs from the onboard accelerometer for rate-adaptive functionality, allowing the system to dynamically adjust pacing rates based on activity levels. The state transitions were updated to include logic for detecting intrinsic cardiac activity in demand and hybrid modes, ensuring precise and safe operation.

To support the serial communication with the DCM, additional modules were designed to handle parameter transmission and verification. The DCM interface was expanded to allow configuration of all programmable parameters, with real-time feedback ensuring accurate synchronization between the doctor's input and the pacemaker's stored values. Additionally, new logic was added to display real-time egram data, improving the system's monitoring capabilities.

Looking ahead, we anticipate extending functionality for dual-chamber pacing modes such as DDD and DDDR, requiring synchronization of atrial and ventricular pacing and maintaining an AV delay. This will necessitate further refinements to the MainStateFlow logic to ensure compatibility with existing safety measures and energy efficiency goals. Throughout these updates, the focus remains on maintaining modularity, optimizing energy consumption, and enhancing patient safety while accommodating new features.

6. Module Clarification

AOO and AOOR 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: In the AOO state machine, the transition from AOO_Charge to Atrial_Pacing is triggered after a fixed interval calculated as LRI - atrial_pulse_width, ensuring a consistent pacing rate based on the configured Lower Rate Interval. In contrast, the AOOR state machine transitions based on an adaptive interval, Adaptive_Interval - atrial_pulse_width, where the interval dynamically adjusts according to the patient's activity level detected by motion sensors. This difference allows AOOR to provide rate-responsive pacing tailored to the patient's metabolic demands, while AOO maintains a steady pacing rate for scenarios requiring constant support.

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.

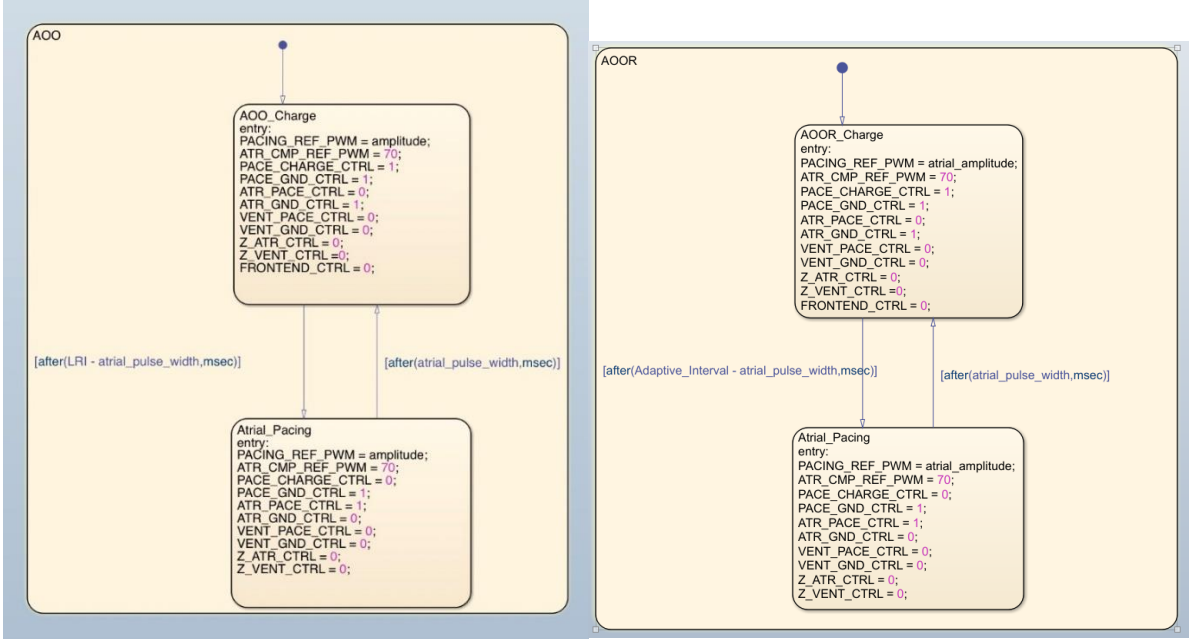


Figure 32. AOO and AOOR Mode State Machine

AAI and AAIR 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: In AAI, the transition from sense to Atrial_Pacing occurs if no intrinsic atrial activity is detected within LRI - ARP, ensuring demand-based pacing. In AAIR, this transition is based on the adaptive interval (Adaptive_Interval - ARP) and only occurs if no intrinsic activity (ATR_CMP_DETECT == 0) is detected. The difference exists because AAIR dynamically adjusts the pacing interval based on activity level to meet metabolic demands, unlike the fixed interval in AAI.

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.

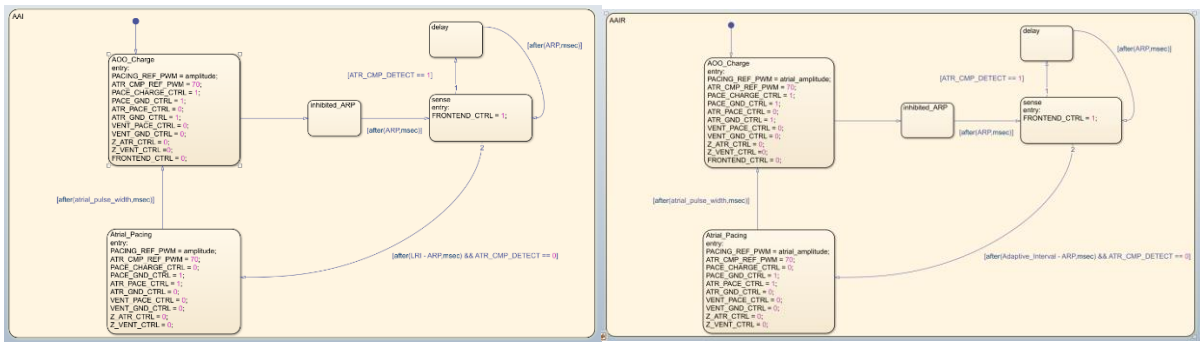


Figure 33. AAI and AAIR Mode State Machine

VOO and VOOR 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: In VOO, the transition from VOO_Charge to Ventricular_Pacing occurs after a fixed interval calculated as LRI - ventricular_pulse_width, ensuring constant-rate ventricular pacing. In VOOR, this transition is based on the adaptive interval, calculated as Adaptive_Interval - ventricular_pulse_width, which dynamically adjusts according to activity levels detected by the motion sensor. The difference arises because VOOR provides rate-responsive pacing to match the patient's metabolic needs during varying activity levels, while VOO maintains a steady pacing rate regardless of activity.

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 34. VOO and VOOR Mode State Machine

VVI and VVIR 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: In VVI, the transition from sense to Ventricular_Pacing occurs if no intrinsic ventricular activity is detected within LRI - VRP, where VRP is the Ventricular Refractory Period, ensuring demand-based pacing. In VVIR, this transition is based on the adaptive interval, calculated as Adaptive_Interval - VRP, and occurs only if no intrinsic activity (VENT_CMP_DETECT == 0) is detected. The difference arises because VVIR dynamically adjusts the pacing interval based on the patient's activity level to meet metabolic demands, while VVI uses a fixed interval for consistent demand-based pacing.

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.

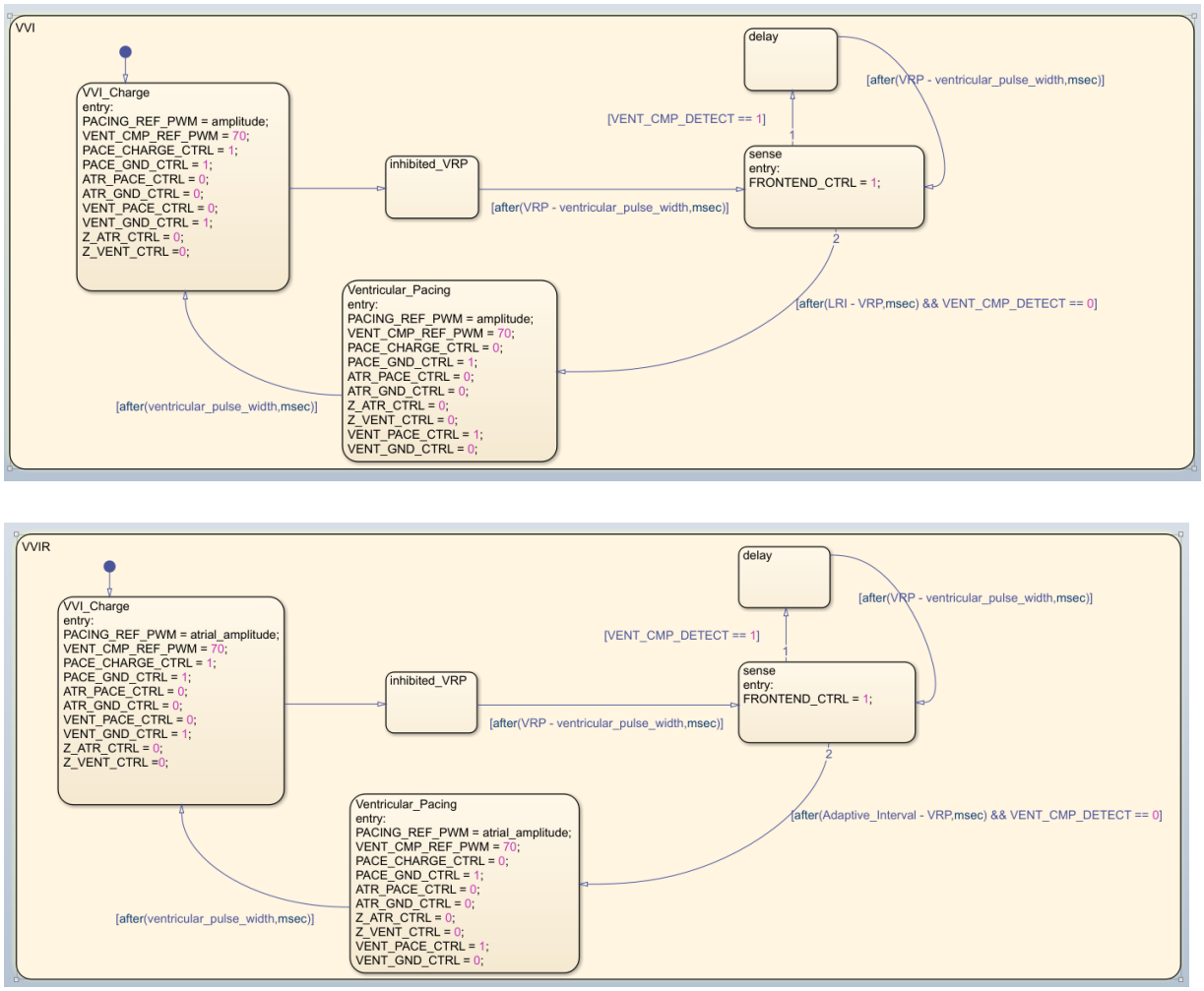


Figure 35. VVI and VVIR Mode State Machine

Device-Controller Monitor

Requirements

1. User Registration System

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 parameter's requirements for assignment 2. P = { Lower Rate Limit, Upper Rate Limit, Atrial Amplitude, Atrial Pulse Width, Ventricular Amplitude, Ventricular Pulse Width, VRP, ARP, Maximum Sensor Rate, Fixed AV Delay, Ventricular Sensitivity, PVARP, Hysteresis, Rate Smoothing, Activity Threshold, Reaction Time, Response Factor, Recovery Time} and for each parameter p ϵ P, the system is required to provide a function or methods that can process the parameter into the specified pacing modes and validating the input value range. Otherwise an error message should be printed to notify the users.

Pacing Mode are specified as M = {AOO, VOO, AAI, VVI, AOOR, VOOR, VVIR, DDD, DDDR}. When pacing mode is selected, the corresponding parameter should be displayed only at a visible position in the program. Any position that can clearly visualized by the user is applicable. For mode m ϵ M and being selected, parameters 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 Transmission

The user can view the electrogram data on the screen or through a printed copy. The user also has the option if selecting which electrograms are viewed and the resolution utilized. The internal electrogram options provided are the following: an atrial internal electrogram option provided, a ventricular internal electrogram option provided, An atrial and ventricular internal electrogram option provided. For the surface ECG, the user can select between the gain utilized and whether the high pass filtering is on. For the internal ECG, the display gain is selectable and applies to all channels.

5. DCM Utility Functions

Implements utility functions to enable access to DCM utilities. One such utility is the "About" function which displays: Application model number, Application software revision number currently in use, DCM serial number, and Institution name. Another one is the "Set Clock" function which sets the date and time of the device. There's also the "New Patient" function which allows a new device to be interrogated without exiting the software application. The last function is the "Quit" function which ends a telemetry session.

6. Printed Reports

Allows the user to print parameter and status reports or bradycardia diagnostic reports with specific header information at the user's request.

The available parameter and status reports are as follows: A Bradycardia Parameters Report, a Temporary Parameters Report, an Implant Data Report, a Threshold Test Results Report, a Measured Data Report, a Marker Legend Report, a Session Net Change Reort, and a Final Report consisting of the Measured Data, Threshold Testing, Histograms, Implement Data, and Net Change Reports

The available bradycardia diagnostic reports are: a Rate Histogram Report and a Trending Report

The header information that must be included are: the institution name, date and time of report printing, device model and serial number, DCM serial number, application model and version number.

7. Programmable Parameters

The programmable parameters as well as their minimum and maximum values are shown in the table below.

Programmable Parameters	Units	Minimum Value	Maximum Value
Lower Rate Limit	bpm	30	175
Upper Rate Limit	bpm	50	175
Max Sensor Rate	bpm	50	175
Fixed AV Delay	ms		
Atrial Amplitude	V	1.0	5.0
Atrial Pulse Width	ms	1	30
Atrial Sensitivity	V	0.0	5.0
Ventricular Amplitude	V	1.0	5.0
Ventricular Pulse Width	ms	1	30
Ventricular Sensitivity	V	0.0	5.0
ARP	ms	150	500
VRP	ms	150	500
PVARP	ms	150	500
Hysteresis	N/A	0	175
Activity Threshold	N/A		
Reaction Time	sec	10	50
Response Factor	N/A	1	16
Recovery Time	min	1	16

8. Serial Communication

Serial communication enables information such as programmable parameters and telemetry data to be transmitted and received between the DCM and the Pacemaker.

Design Decisions – Assignment 1

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

Design Decisions – Assignment 2

For assignment 2, the software model of the pacemaker project has a build a more comprehensive structure for the GUI and control system. The design decision of the DCM is divided up into 7 parts. The design matrix below illustrates the updated features and original functionalities of the entire DCM design.

Category	Design Decision	Justification
User Management	1. As continued using the test files for username and passwords storage management 2. PyQt6 based debugging message box for invalidate input type and length	1. Based on 10 users' limitation, a local and simple credentials storage method is applicable. 2. Enhances user experiences and better and simple guide for user interface.
Parameter Management	1. ParameterManager.py file include the class for all programable parameters getter and setter functions. 2. Nominal values for each parameter are instance for the parameter manager class. 3. Validation of parameter input value range in application window class. 4. Parameters value can be modified by user in application window using PyQt6 build in QLineEdit() class and the stored and accessed within users.dat data file using pickle method.	1. A centralised management of all programmable parameters. 2. Setting of default values will be shown in the user interface for simple instructions of input value range. 3. Safety insurance for input parameter value range prevent over pacing or non-sufficient pacing. 4. Simplifies persistence and retrieval of personalized user settings.
Pacing Mode Selection	1. Drop down menu for rate smoothing and activity threshold 2. Dynamically update visible parameter fields based on the selected mode. 3. Map pacing modes to numeric values for device communication.	1. Provides an intuitive and error-free way to choose desired values for fix parameters choices. 2. Reduces cognitive load and prevents accidental parameter changes. 3. Simplifies serial communication by using numeric representations.
Serial Communication	1. Use pyserial library for device communication. 2. Validate responses and data length check for send and received data packets. 3. Feedback check for using any stored parameters for assigned user. 4. Multi-platform adaptive: Both MacOS and Windows system supported for the pacemaker device.	1. Reliable serial communication for embedded system. 2. Easier debugging process for DCM application, 3. Allows testing individually without hardware. (ie. Egram data will show a default plot with serial connection)

Egram Plotting	1. Use matplotlib for real-time visualization of atrium and ventricle signals. 2. Allow user to choose atrium and ventricle signal for display 3. Button assignment for start and stop telemetry.	1. Provides a comprehensive display of pacing data plots. 2. Provide user with a simple control over display for different diagnostic needs. 3. Prevent timeout for serial connection.
Error Handling	1. Validate user inputs with error messages. 2. Catch and handle exceptions: SerialException, IndexError	1. Enhances robustness and maintains device safety. 2. Prevents crashes in unexpected scenarios, ensuring system stability.
UI Design	1. Basic functionalities continue with previous design. 2. Added new parameters for display and edit: Maximum Sensor Rate, Fixed AV Delay Ventricular Sensitivity PVARP Hysteresis Rate Smoothing Activity Threshold Reaction Time Response Factor Recovery Time 3. Added new pacing modes in the drop-down menu: AOOR VOOR AAIR VVIR DDD DDDR	1. Provides a modern, responsive framework for desktop applications. 2. The original clean display and usage guidelines have been maintained. Also added a new pacing mode and corresponding parameters.

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 set correctly in default and changed 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.

4. Real-Time Electrogram Plotting

Validation: Ensure that atrial and ventricular signals are displayed in real-time with accurate axes labels, synchronized time updates, and correct signal ranges (-6V to +6V). Verify that toggling visibility for atrial and ventricular plots works as intended.

Verification: Test with simulated and real pacemaker data to confirm real-time updates in the plots. Toggle atrial and ventricular plots to ensure visibility states update dynamically.

Simulate disconnected devices to validate fallback functionality with random signal generation.

Inject edge-case data (e.g., out-of-range signals) and error exception handling without crashes.

5. Parameter Management

Validation: Ensure all parameters limited in appropriate ranges and support configurations like "OFF" values for atrial and ventricular amplitude. Confirm proper serialization and deserialization of user-specific parameters using pickle.

Verification: Use unit tests to validate getter and setter methods for all parameters.

Test edge cases by providing out-of-range inputs and confirm appropriate error handling (e.g., raising exceptions).

Validate "OFF" state functionality by setting applicable parameters drop-down menu to "OFF" and checking their internal values are stored as 0.

Save and reload parameters for multiple users, verifying data integrity and correctness.

6. Serial Communication with Pacemaker Device

Validation: Ensure reliable communication with the pacemaker by correctly formatting and parsing data packets, handling timeouts, and validating the integrity of received data. Verify real-time parameter feedback by comparing sent and received values.

Verification: Open and close serial ports, ensuring proper transitions and error handling for invalid ports. Send and receive formatted packets, verifying the data type and length to expected formats 3B13fH6f according to the MATLAB module.

Compare sent parameter values with received feedback from the pacemaker to confirm synchronization and accuracy.

Test and Result

User Login/Registration/deletion functionality

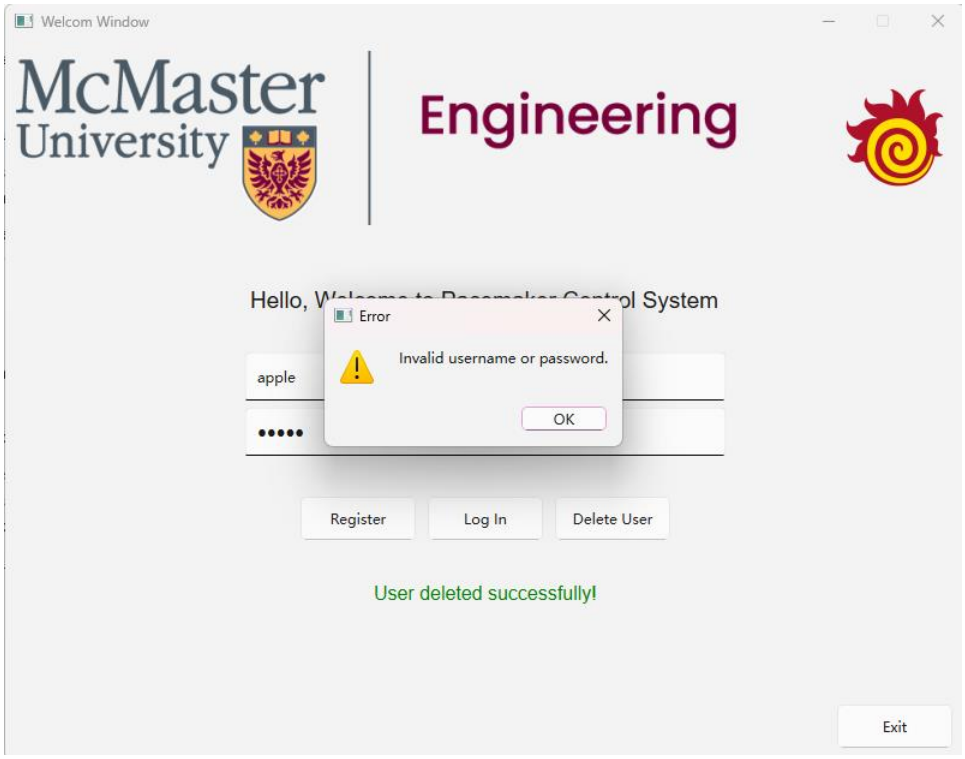


Figure 36 Testing unknown user registration

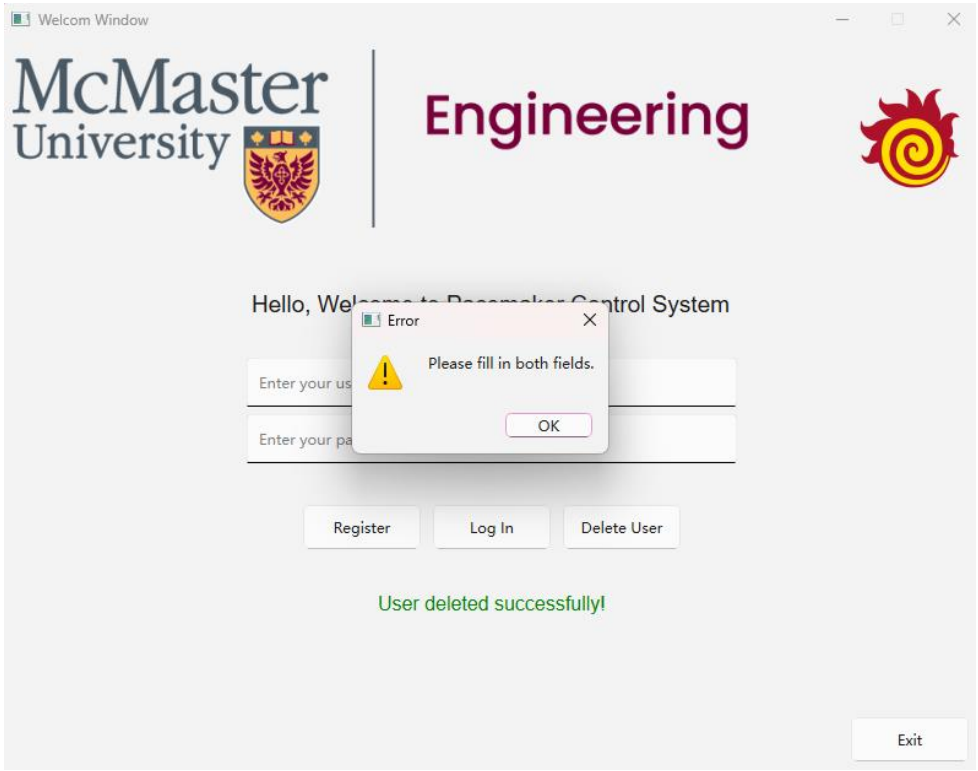


Figure 37 Testing unknown input

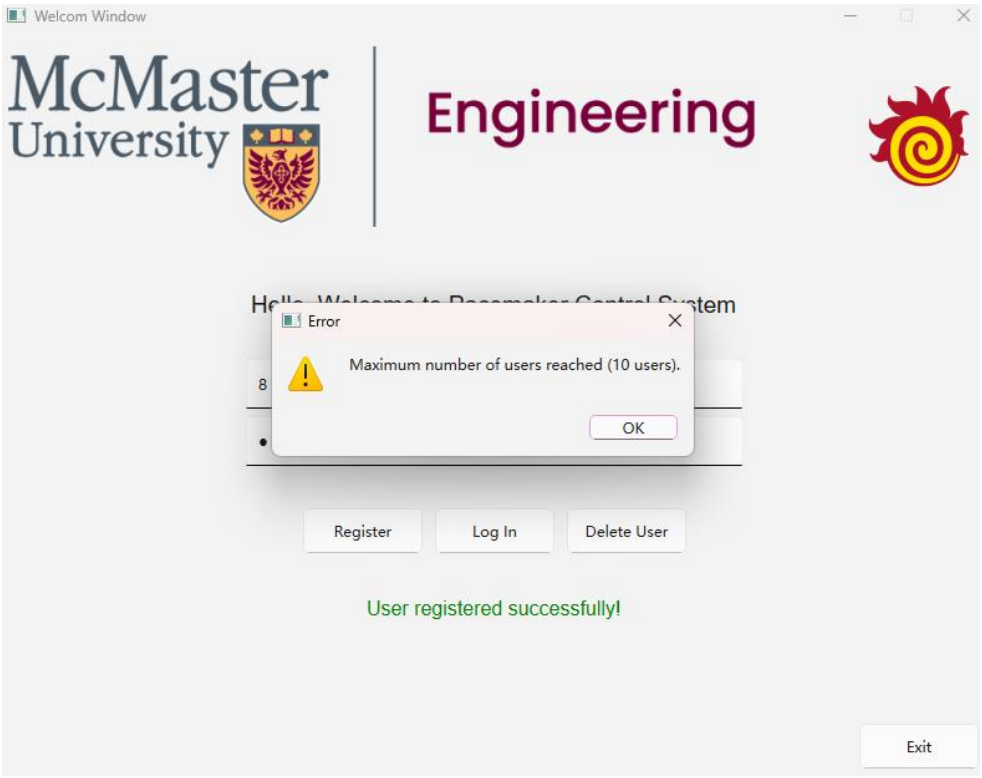


Figure 38 Testing for Maximum User

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

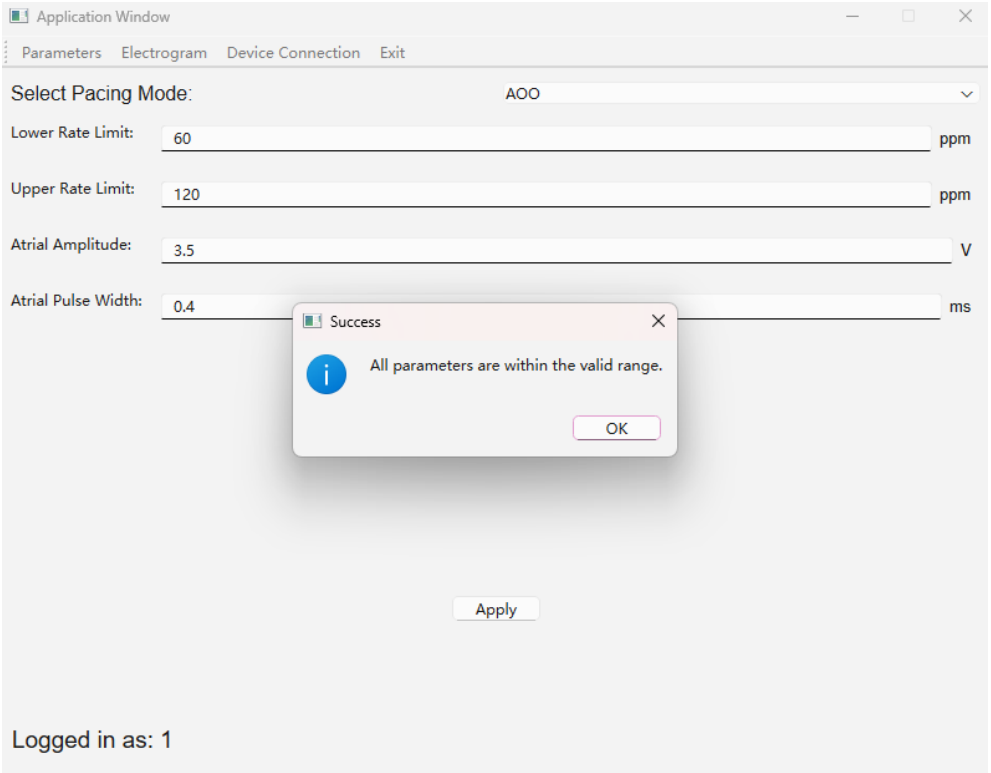


Figure 39 Testing for Valid Input of programmable parameter

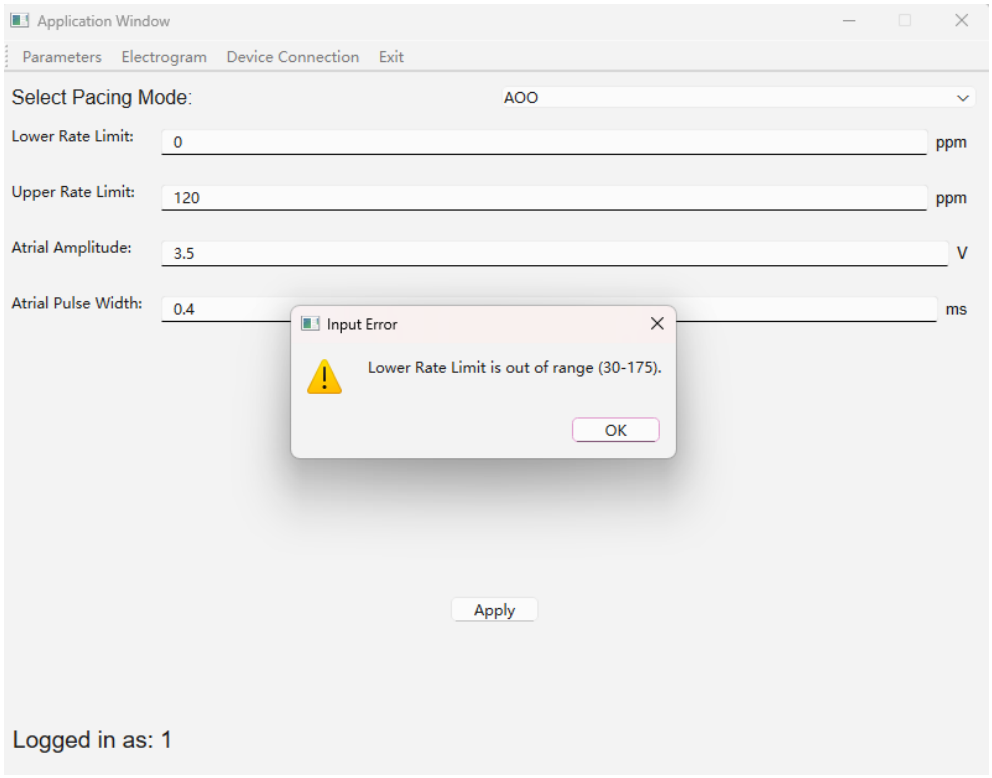


Figure 40 Testing for Invalid input parameter LRL

Pacing Mode Selection and Real-Time Display

The current pacing mode is selected as AOO:

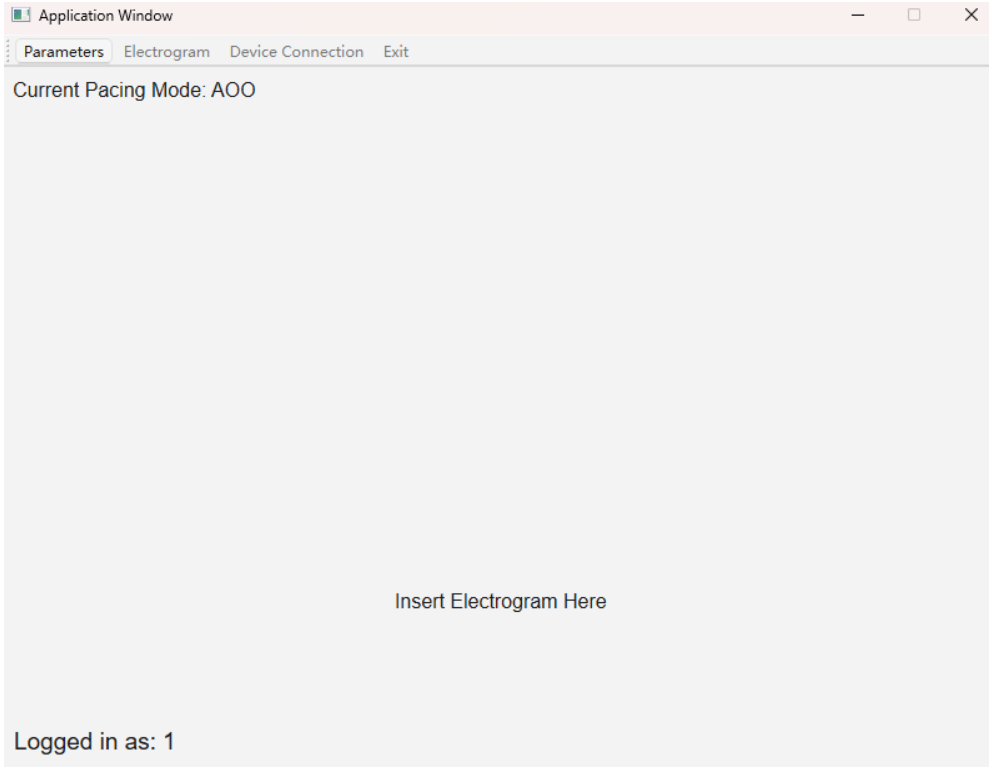


Figure 41 Testing for Electrogram Real-Time Display of Pacing Mode

Parameter storage and access with assigned “Apply” button: PASS

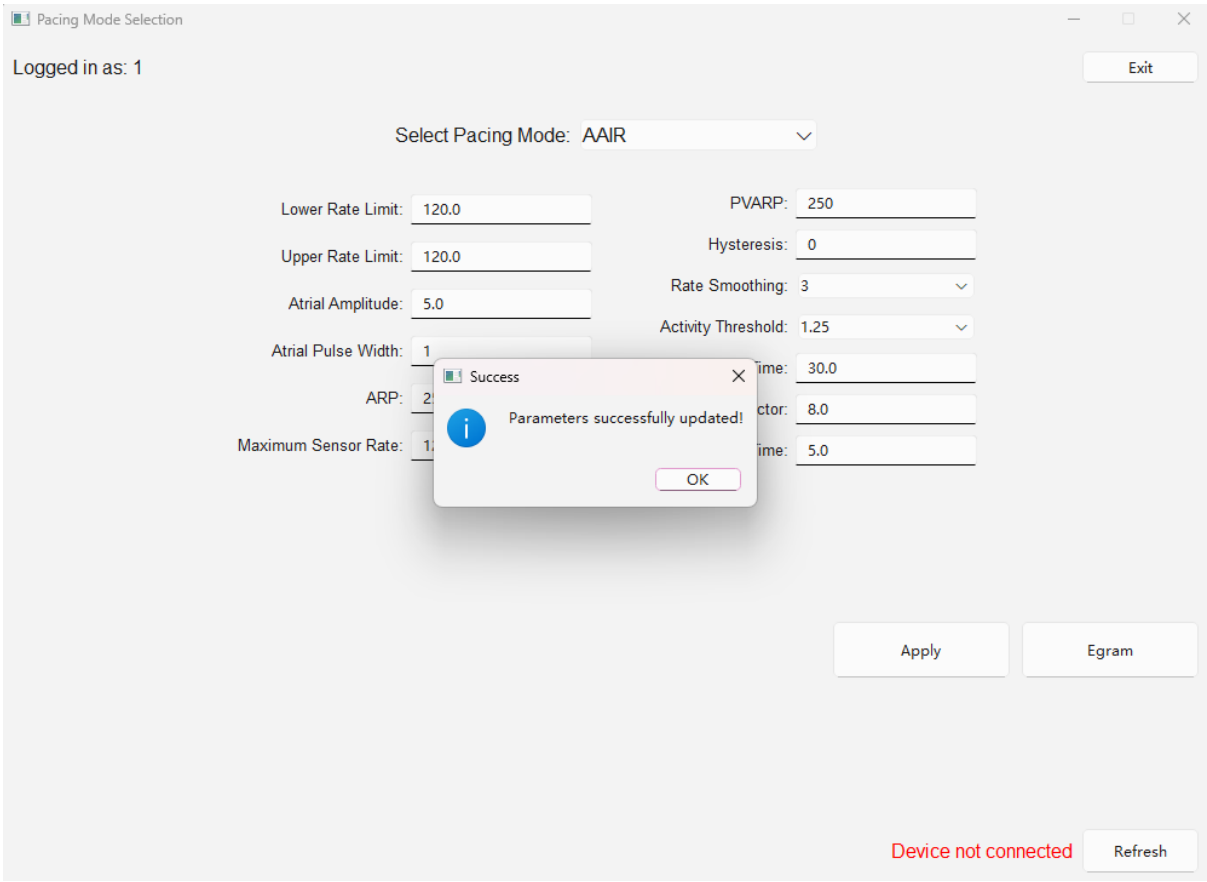


Figure 42 Apply parameter valus into the data file feedaback message

```
User: 1, Parameters: {'_ParameterManager__lrl': 120.0, '_ParameterManager__url': 120.0, '_ParameterManager__msr': 120.0, '_ParameterManager__fixed_av_delay': 150, '_ParameterManager__atrial_amplitude': 5.0, '_ParameterManager__ventricular_amplitude': 5.0, '_ParameterManager__atrial_pulse_width': 1, '_ParameterManager__ventricular_pulse_width': 1, '_ParameterManager__atrial_sensitivity': 0.1, '_ParameterManager__ventricular_sensitivity': 0.1, '_ParameterManager__arp': 250.0, '_ParameterManager__vrp': 320, '_ParameterManager__pvarp': 250, '_ParameterManager__hysteresis': 0, '_ParameterManager__rate_smoothing': 3, '_ParameterManager__activity_threshold': 1.25, '_ParameterManager__reaction_time': 30.0, '_ParameterManager__response_factor': 8.0, '_ParameterManager__recovery_time': 300.0}
```

Figure 43 Debug message for datas stored in the users.dat file

Serial connection feedback message: PASS

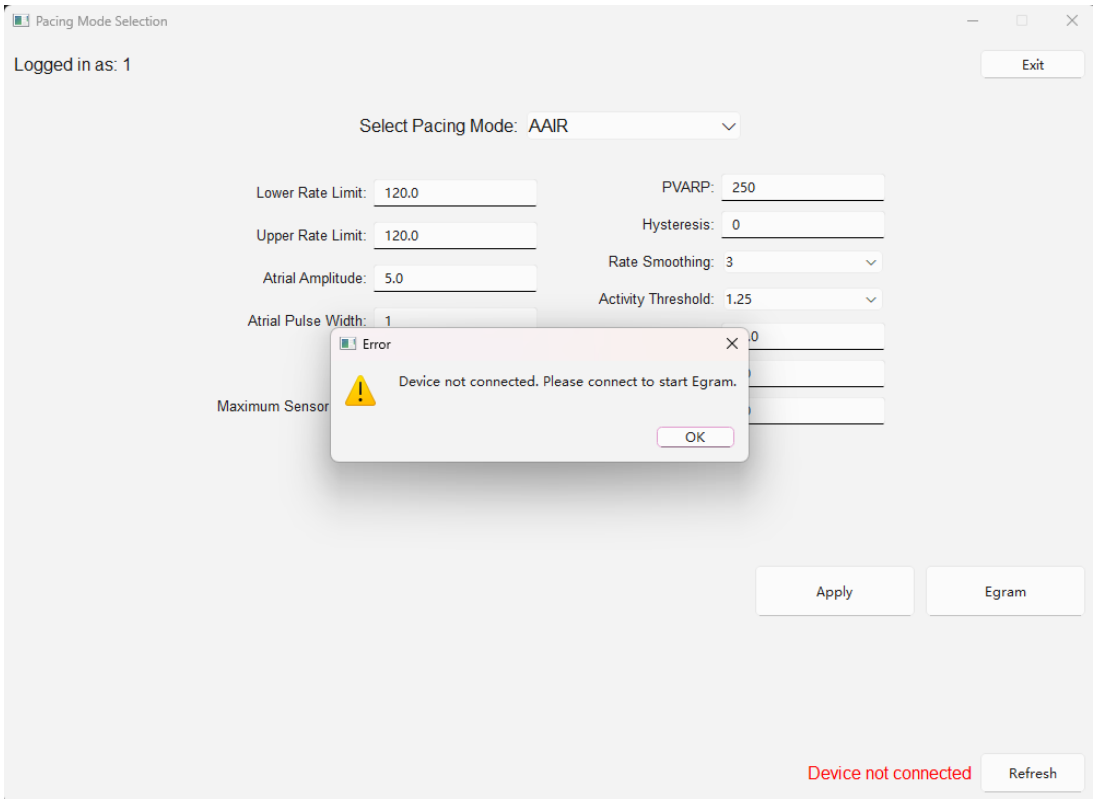


Figure 44 Device not connected warning for using "Egram" button to generate plot

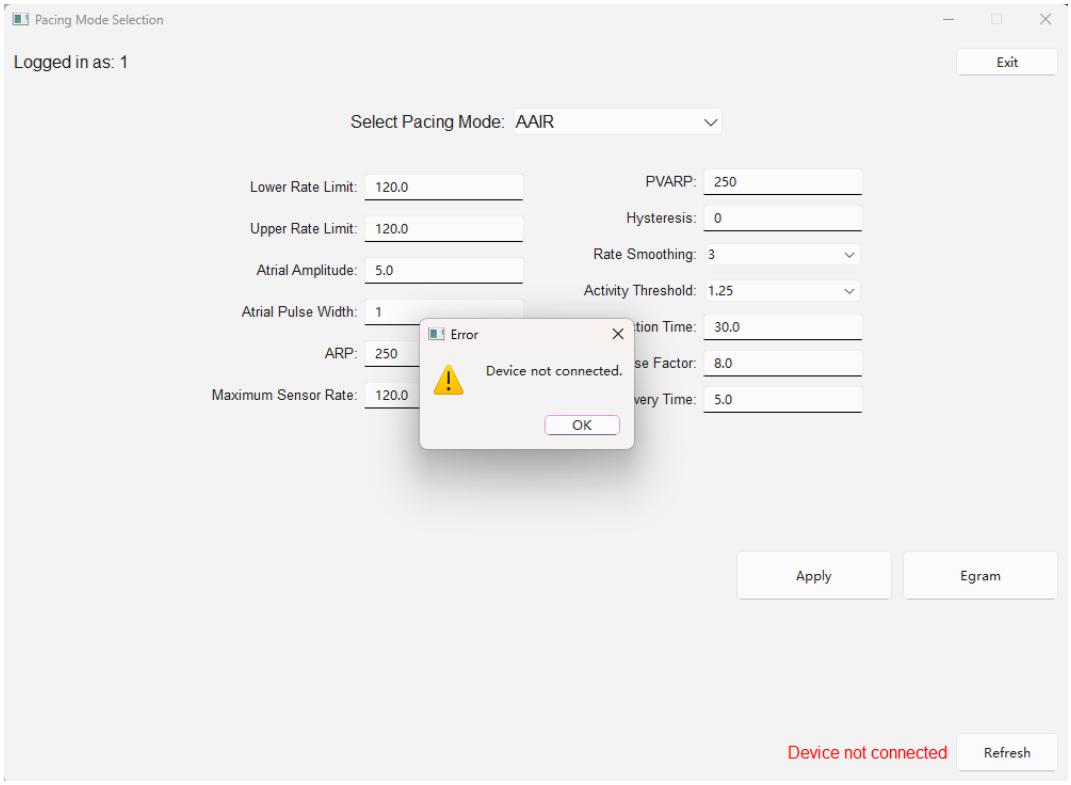


Figure 45 Device not connected warning for using "Apply" button to save parameters

Electrogram Plot Functionality Test : PASS

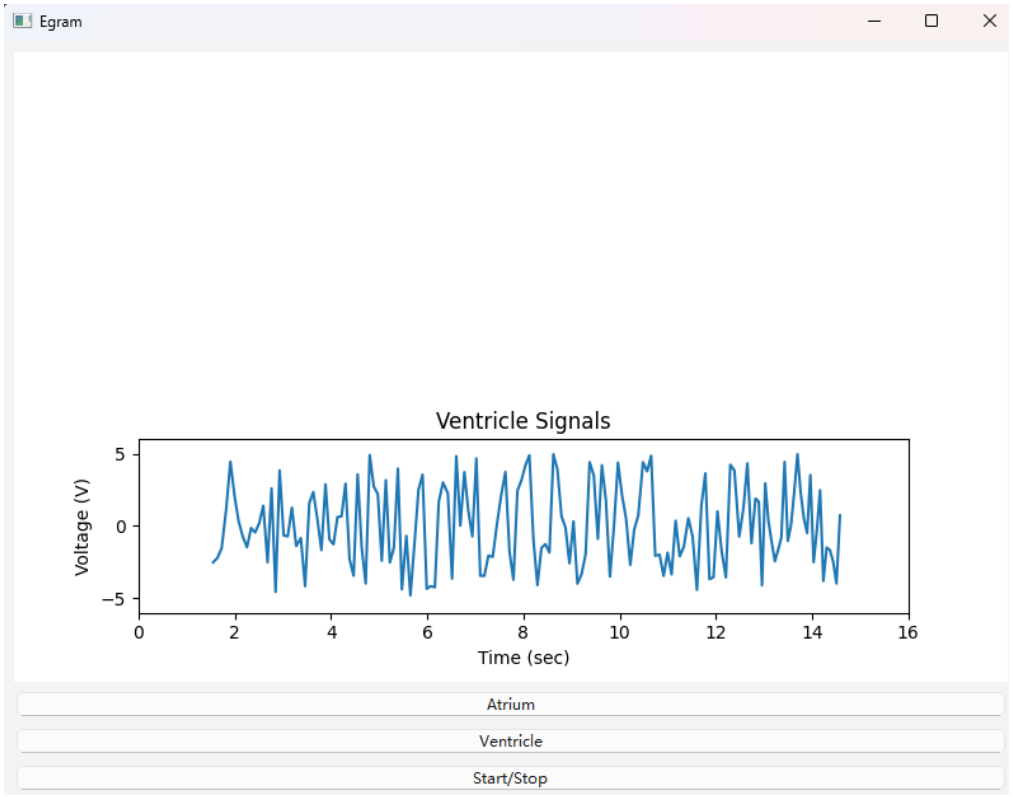


Figure 46 Button test for only showing "Ventricle Signals"

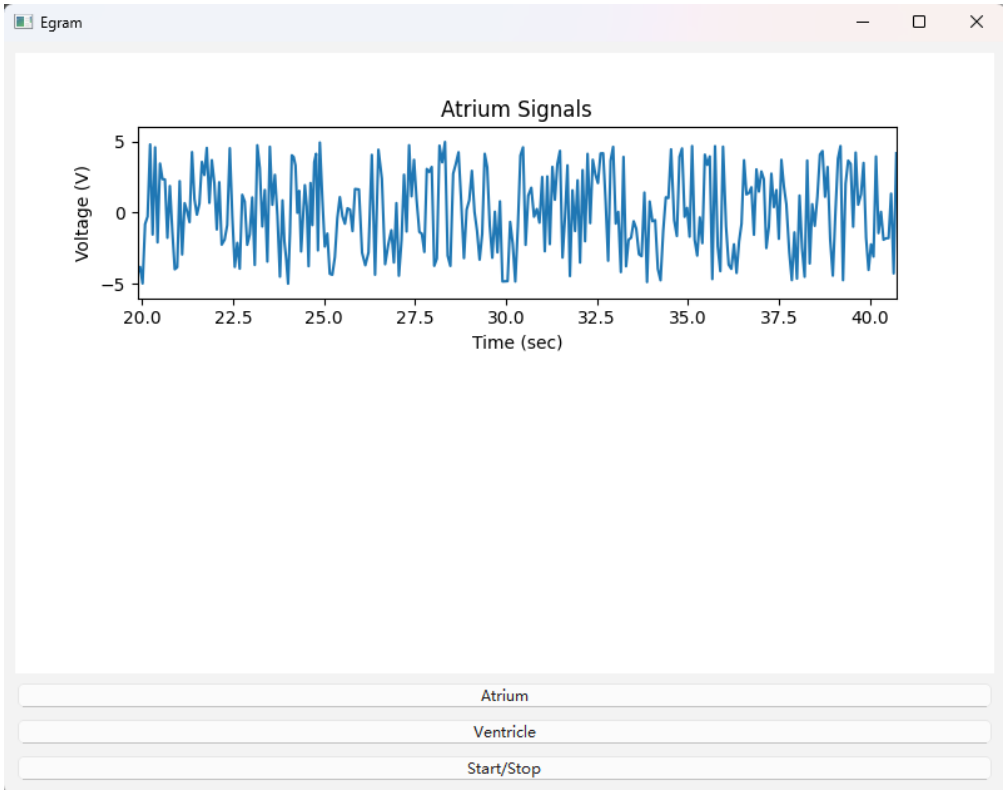


Figure 47Button test for only showing "Atrium Signals"

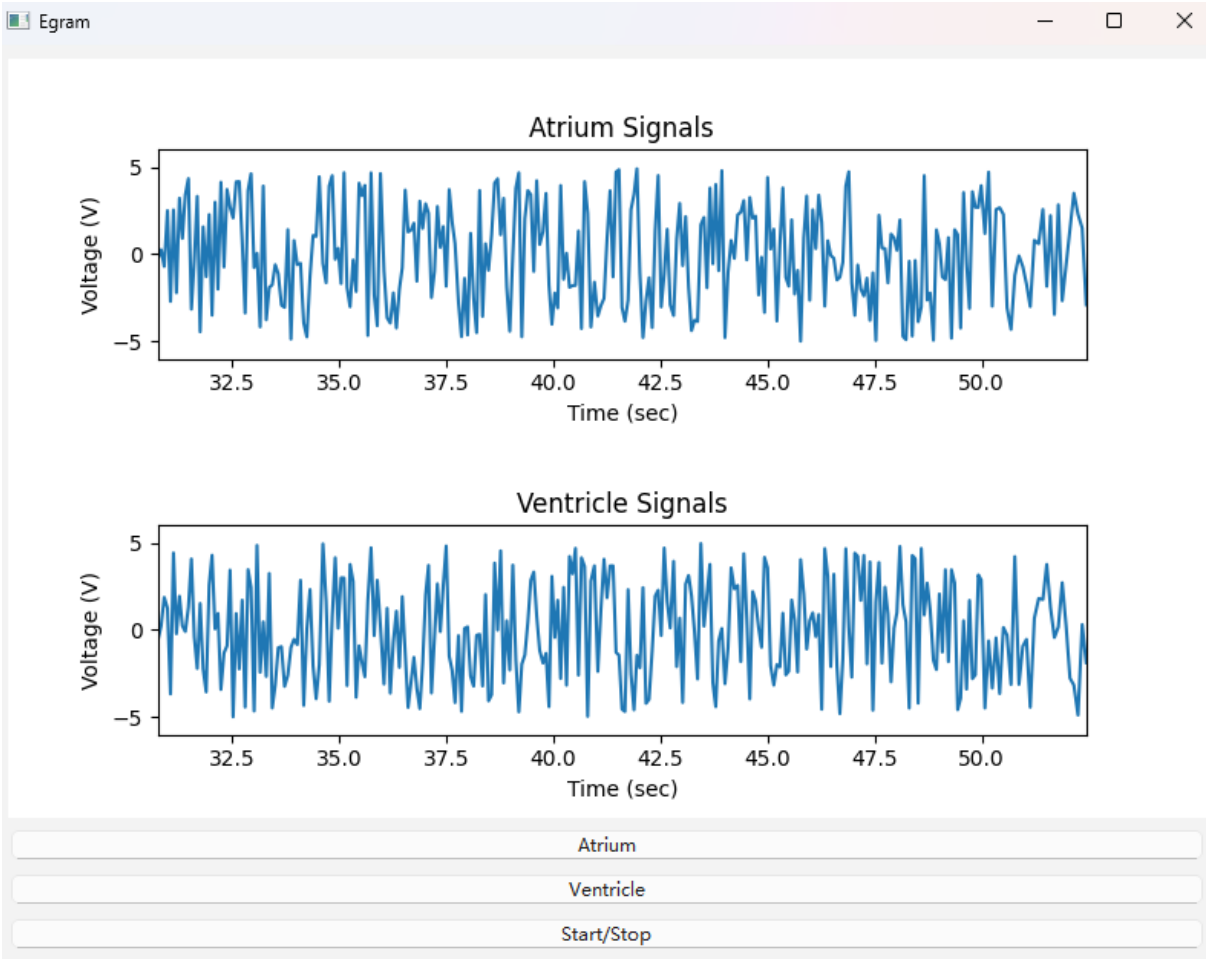


Figure 48 Test for start/stop telemetry and time shifting

Requirement Changes

3. 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 parameters listed on the entire project requirements. 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.

4. 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.

5. 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.

6. 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 hardware connecting with the board.

7. 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.

8. Pacing mode DDD / DDDR for bonus

Current state: All required pacing modes except DDD and DDDR are implemented beyond the project requirements from DCM end. And the corresponding parameters are also initialized for user to modify and save.

Expected Change: Bonus pacing mode DDD and DDDR are only implemented in the UI design but not actually functional. In the future implementation, the parameters: Dynamic AV delay, Sensed AV Delay offset, PVARP Extension, ATR Duration, ATR fallback mode, ATR fallback time will be added too.

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 protocol 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.

Electrogram Visualization

Transition to more advanced visualization frameworks (e.g., PyQtGraph or real-time 3D plotting libraries) for better performance and interactivity.

Enhanced real-time performance for higher sampling rates.

Improved user experience with additional features like zooming, panning, or overlaying multiple signal traces.

DCM Code

1. MainWindow.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)
        # 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()

        # 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)
        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
                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.")

    def open_application_window(self, username):
        # 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()
```

2. ApplicationWindow.py

```
import serial
import serial.tools.list_ports
from PyQt6.QtCore import *
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
import pickle
from ParameterManager import ParameterManager
import struct
import time
from egram_plot import EgramPlot

# Application window for pacemaker
class ApplicationWindow(QMainWindow):
    def __init__(self, username):
        super().__init__()

        # Create an instance of ParameterManager
        self.parameter_manager = ParameterManager()

        # Initialize for user file to store all parameters
        self.username = username
        self.users_file = 'users.dat'
        self.load_user_parameters()

        # Create an instance of ParameterManager for the current user
        self.parameter_manager = self.user_parameters.get(username,
ParameterManager())

        # Set up window title and size
        self.setWindowTitle("Pacing Mode Selection")
        self.setFixedSize(QSize(1000, 700))

        # Store the connected port
        self.connected_port = None

        # 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(),
            "Maximum Sensor Rate": QLineEdit(),
            "Fixed AV Delay": QLineEdit(),
            "Ventricular Sensitivity": QLineEdit(),
            "PVARP": QLineEdit(),
            "Hysteresis": QLineEdit(),
            "Rate Smoothing": QComboBox(),
            "Activity Threshold": QComboBox(),
            "Reaction Time": QLineEdit(),
            "Response Factor": QLineEdit(),
            "Recovery Time": QLineEdit(),
        }

        # Set initial values in the fields using the ParameterManager
        for field_name, field_widget in self.fields.items():
            if isinstance(field_widget, QComboBox):
                if field_name == "Rate Smoothing":
                    field_widget.addItems(["0", "3", "6", "9", "12", "15", "18",
"21", "25"])
                elif field_name == "Activity Threshold":
                    field_widget.addItems([
                        "1.13", "1.25", "1.4", "1.6", "2", "2.4", "3"
                    ])
                current_value = str(getattr(self.parameter_manager,
f"get{field_name.replace(' ', '')}")())
                index = field_widget.findText(current_value,
Qt.MatchFlag.MatchContains)
                field_widget.setCurrentIndex(index if index != -1 else 0)
            else:
                field_widget.setText(str(getattr(self.parameter_manager,
f"get{field_name.replace(' ', '')}")()))

        # Create the main layout for pacing mode selection
        main_layout = QVBoxLayout()

        # Header layout for username display and exit button
        header_layout = QHBoxLayout()

        # Username display
        self.username_label = QLabel(f"Logged in as: {username}")
        self.username_label.setFont(QFont('Arial', 12))
        header_layout.addWidget(self.username_label,
alignment=Qt.AlignmentFlag.AlignLeft)
```

```
        # Exit Button
        self.exit_button = QPushButton("Exit")
        self.exit_button.setFixedSize(100, 30)
        self.exit_button.clicked.connect(self.exit_program)
        header_layout.addStretch()  # stretch exit button to the right
        header_layout.addWidget(self.exit_button)

        # Add header layout to the main layout
        main_layout.addLayout(header_layout)

        # Spacer for better display
        main_layout.addSpacing(20)

        # Pacing Mode Selection Dropdown
        pacing_mode_layout = QHBoxLayout()
        pacing_mode_label = QLabel("Select Pacing Mode:")
        pacing_mode_label.setFont(QFont('Arial', 12))

        self.pacing_mode_combo = QComboBox()
        self.pacing_mode_combo.setFont(QFont('Arial', 12))  # Combo box for drop
down menu
        self.pacing_mode_combo.setFixedSize(200, 30)
        # All pacing modes
        self.pacing_mode_combo.addItems(["None", "AOO", "VOO", "AAI", "VVI", "AOOR",
"AAIR", "VOOR", "VVIR", "DDD", "DDDR"])
        self.pacing_mode_combo.currentIndexChanged.connect(self.update_parameters)

        pacing_mode_layout.addStretch()
        pacing_mode_layout.addWidget(pacing_mode_label)
        pacing_mode_layout.addWidget(self.pacing_mode_combo)
        pacing_mode_layout.addStretch()

        # Add pacing mode layout to the main layout
        main_layout.addLayout(pacing_mode_layout)

        # Spacer for better display
        main_layout.addSpacing(20)

        # Parameter Fields
        self.field_containers = {}

        # Create a main horizontal layout for two columns
        param_main_layout = QHBoxLayout()
```

```
        left_param_layout = QVBoxLayout()
        right_param_layout = QVBoxLayout()

        # Add fields for parameter display
        for i, (field_name, field_widget) in enumerate(self.fields.items()):
            label = QLabel(f"{field_name}:")
            label.setFont(QFont('Arial', 10))
            field_widget.setFixedSize(150, 25)

            self.field_containers[field_name] = (label, field_widget)

            field_layout = QHBoxLayout()
            field_layout.addWidget(label, alignment=Qt.AlignmentFlag.AlignRight)
            field_layout.addWidget(field_widget,
alignment=Qt.AlignmentFlag.AlignLeft)

            if i < 9:
                left_param_layout.addLayout(field_layout)
            else:
                right_param_layout.addLayout(field_layout)

        # Add to the main horizontal layout
        param_main_layout.addStretch()
        param_main_layout.addLayout(left_param_layout)
        param_main_layout.addSpacing(50)
        param_main_layout.addLayout(right_param_layout)
        param_main_layout.addStretch()

        # Add the parameter layout to the main layout
        main_layout.addLayout(param_main_layout)

        # Spacer for better display
        main_layout.addSpacing(20)

        # Layout for Apply Button
        button_layout = QHBoxLayout()

        # Apply Button for Parameter
        self.apply_button = QPushButton("Apply")
        self.apply_button.setFixedSize(150, 50)
        self.apply_button.clicked.connect(self.apply_parameters)
        button_layout.addStretch()
        button_layout.addWidget(self.apply_button,
alignment=Qt.AlignmentFlag.AlignCenter)
```

```
        # Add button layout to main layout
        main_layout.addLayout(button_layout)

        # Spacer for better display
        main_layout.addSpacing(20)

        # Egram Button
        self.egram_button = QPushButton("Egram")
        self.egram_button.setFixedSize(150, 50)
        self.egram_button.clicked.connect(self.start_egram)
        button_layout.addWidget(self.egram_button,
alignment=Qt.AlignmentFlag.AlignCenter)

        # Connection Status Check Button
        connection_layout = QHBoxLayout()
        self.status_msg = QLabel("Device not connected")
        self.status_msg.setFont(QFont('Arial', 12))
        self.status_msg.setStyleSheet("color: red;")
        self.refresh_button = QPushButton("Refresh")
        self.refresh_button.setFixedSize(100, 40)
        self.refresh_button.clicked.connect(self.check_connection_status)

        connection_layout.addStretch()
        connection_layout.addWidget(self.status_msg,
alignment=Qt.AlignmentFlag.AlignRight)
        connection_layout.addWidget(self.refresh_button,
alignment=Qt.AlignmentFlag.AlignRight)

        main_layout.addLayout(connection_layout)

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

    def load_user_parameters(self):
        try:
            with open(self.users_file, 'rb') as file:
                self.user_parameters = pickle.load(file)
                print("Loaded user parameters:")
                for user, params in self.user_parameters.items():
                    print(f"User: {user}, Parameters: {params.__dict__}")
        except FileNotFoundError:
```

```
            self.user_parameters = {}
            print("User data file not found. Creating new data structure.")

    def save_user_parameters(self):
        self.user_parameters[self.username] = self.parameter_manager
        with open(self.users_file, 'wb') as file:
            pickle.dump(self.user_parameters, file)
        print("Saved user parameters:")
        for user, params in self.user_parameters.items():
            print(f"User: {user}, Parameters: {params.__dict__}")

    # Method to get relevant parameters for the selected pacing mode
    def get_relevant_parameters_for_mode(self, mode):
        relevant_params_dict = {
            "AOO": ["Lower Rate Limit", "Upper Rate Limit", "Atrial Amplitude",
"Atrial Pulse Width"],
            "AAI": ["Lower Rate Limit", "Upper Rate Limit", "Atrial Amplitude",
"Atrial Pulse Width", "Atrial Sensitivity", "ARP", "PVARP", "Hysteresis", "Rate
Smoothing"],
            "VOO": ["Lower Rate Limit", "Upper Rate Limit", "Ventricular Amplitude",
"Ventricular Pulse Width"],
            "VVI": ["Lower Rate Limit", "Upper Rate Limit", "Ventricular Amplitude",
"Ventricular Pulse Width", "Ventricular Sensitivity", "VRP", "Hysteresis", "Rate
Smoothing"],
            "AOOR": ["Lower Rate Limit", "Upper Rate Limit", "Maximum Sensor Rate",
"Atrial Amplitude", "Atrial Pulse Width", "Activity Threshold", "Reaction Time",
"Response Factor", "Recovery Time"],
            "AAIR": ["Lower Rate Limit", "Upper Rate Limit", "Maximum Sensor Rate",
"Atrial Amplitude", "Atrial Pulse Width", "Atrial Sensitivity", "ARP", "PVARP",
"Hysteresis", "Rate Smoothing", "Activity Threshold", "Reaction Time", "Response
Factor", "Recovery Time"],
            "VOOR": ["Lower Rate Limit", "Upper Rate Limit", "Maximum Sensor Rate",
"Ventricular Amplitude", "Ventricular Pulse Width", "Activity Threshold", "Reaction
Time", "Response Factor", "Recovery Time"],
            "VVIR": ["Lower Rate Limit", "Upper Rate Limit", "Maximum Sensor Rate",
"Ventricular Amplitude", "Ventricular Pulse Width", "Ventricular Sensitivity",
"VRP", "Hysteresis", "Rate Smoothing"],
        }
        return relevant_params_dict.get(mode, [])

    # Method to map the pacing mode
    def get_mode_value(self, mode):
        mode_mapping = {
            "AOO": 1,
```

```
            "VOO": 2,
            "AAI": 3,
            "VVI": 4,
            "AOOR": 5,
            "VOOR": 6,
            "AAIR": 7,
            "VVIR": 8
        }
        return mode_mapping.get(mode, 0)  # Default to 0 if "None" or unrecognized

    # Method to apply parameters to ParameterManager
    def apply_parameters(self):
        mode = self.pacing_mode_combo.currentText()
        relevant_params = self.get_relevant_parameters_for_mode(mode)

        errors = 0
        error_text = ""

        # Apply only relevant parameters depending on the pacing modes
        for field_name, field_widget in self.fields.items():
            if field_name in relevant_params:
                if isinstance(field_widget, QComboBox):
                    value = field_widget.currentText().split()[0]
                else:
                    value = field_widget.text()

                if value:
                    try:
                        set_method = getattr(self.parameter_manager,
f"set{field_name.replace(' ', '')}")
                        if field_name in ["Lower Rate Limit", "Upper Rate Limit",
"Maximum Sensor Rate", "Fixed AV Delay", "ARP", "VRP", "PVARP", "Reaction Time",
"Response Factor", "Recovery Time"]:
                            set_method(float(value))
                        elif field_name in ["Atrial Amplitude", "Ventricular
Amplitude", "Atrial Sensitivity", "Ventricular Sensitivity", "Activity Threshold"]:
                            set_method(float(value))
                        elif field_name == "Rate Smoothing":
                            set_method(int(value))
                        elif field_name == "Hysteresis":
                            set_method(int(float(value)))
                    except TypeError:
                        errors += 1
```

```
                        error_text += f"{field_name}: Please use numeric value
input!."
                    except IndexError:
                        errors += 1
                        if field_name == "Lower Rate Limit":
                            error_text += f"{field_name}: Must be between 30 and 175
bpm."
                        elif field_name == "Upper Rate Limit":
                            error_text += f"{field_name}: Must be between 50 and 175
bpm and greater than Lower Rate Limit."
                        elif field_name == "Maximum Sensor Rate":
                            error_text += f"{field_name}: Must be between 50 and 175
bpm and greater than Lower Rate Limit."
                        elif field_name == "Atrial Amplitude" or field_name ==
"Ventricular Amplitude":
                            error_text += f"{field_name}: Must be between 0.1 and 5.0
V."
                        elif field_name == "Atrial Pulse Width" or field_name ==
"Ventricular Pulse Width":
                            error_text += f"{field_name}: Must be between 1 and 30
ms."
                        elif field_name == "Atrial Sensitivity" or field_name ==
"Ventricular Sensitivity":
                            error_text += f"{field_name}: Must be between 0.0 and 5.0
V."
                        elif field_name == "ARP" or field_name == "VRP" or field_name
== "PVARP":
                            error_text += f"{field_name}: Must be between 150 and 500
ms."
                        elif field_name == "Hysteresis":
                            error_text += f"{field_name}: Must be between 0 and 175."
                        elif field_name == "Reaction Time":
                            error_text += f"{field_name}: Must be between 10 and 50
seconds."
                        elif field_name == "Response Factor":
                            error_text += f"{field_name}: Must be between 1 and 16."
                        elif field_name == "Recovery Time":
                            error_text += f"{field_name}: Must be between 1 and 16
minutes."
                        else:
                            error_text += f"{field_name}: Value out of range, please
enter a valid value."

        if errors > 0:
```

```
            QMessageBox.warning(self, "Input Error", f"There are {errors}
error(s):{error_text}")
        else:
            self.save_user_parameters()
            QMessageBox.information(self, "Success", "Parameters successfully
updated!")
            # Send inputted data to the pacemaker device
            self.send_and_validate_parameters()

    # Method to update parameters for the selected pacing mode
    def update_parameters(self):
        mode = self.pacing_mode_combo.currentText()
        self.current_pacing_mode = mode

        # Define visibility for each pacing mode
        visibility_dict = {field: field in
self.get_relevant_parameters_for_mode(mode) for field in self.fields}
        self.set_field_visibility(visibility_dict)

    # Method to set visibility of parameter fields
    def set_field_visibility(self, visibility_dict):
        for field_name, is_visible in visibility_dict.items():
            label, field_widget = self.field_containers.get(field_name, (None, None))
            if label and field_widget:
                label.setVisible(is_visible)
                field_widget.setVisible(is_visible)

    # Method to check connection status
    def check_connection_status(self):

        ports = serial.tools.list_ports.comports()
        available_ports = [port.device for port in ports]

        # Common ports for macOS and Windows
        common_ports = ["COM9", "COM8", "COM7", "COM6", "COM5", "COM4", "COM3"]  #
Windows
        mac_ports = ["/dev/tty.usbserial", "/dev/tty.usbmodem", "/dev/cu.usbserial",
"/dev/cu.usbmodem"]  # macOS

        # Add macOS ports to the list of ports to check
        all_ports_to_check = available_ports + common_ports + mac_ports

        for port in all_ports_to_check:
            try:
```

```
                ser = serial.Serial(port=port, baudrate=115200, timeout=1)
                ser.close()
                self.connected_port = port
                self.status_msg.setText(f"Connected device: {port}")
                self.status_msg.setStyleSheet("color: green;")
                return
            except (serial.SerialException, FileNotFoundError):
                continue
        self.connected_port = None
        self.status_msg.setText("Device not connected")
        self.status_msg.setStyleSheet("color: red;")

    # Method to send parameters via serial communication and validate the response
    def send_and_validate_parameters(self):
        ser = None
        try:
            if not self.connected_port:
                QMessageBox.warning(self, "Error", "Device not connected.")
                return
            # Open the serial port
            ser = serial.Serial(port=self.connected_port, baudrate=115200, timeout=1)

            # Define the header and structure format for sending data
            data_format = '<3B13fH6f'

            mode = self.get_mode_value(self.pacing_mode_combo.currentText())
            lrl = self.parameter_manager.getLowerRateLimit()
            url = self.parameter_manager.getUpperRateLimit()
            msr = self.parameter_manager.getMaximumSensorRate()
            aa = self.parameter_manager.getAtrialAmplitude()
            va = self.parameter_manager.getVentricularAmplitude()
            apw = self.parameter_manager.getAtrialPulseWidth()
            vpw = self.parameter_manager.getVentricularPulseWidth()
            asens = self.parameter_manager.getAtrialSensitivity()
            vsens = self.parameter_manager.getVentricularSensitivity()
            arp = self.parameter_manager.getARP()
            vrp = self.parameter_manager.getVRP()
            pvarp = self.parameter_manager.getPVARP()
            act_thresh = self.parameter_manager.getActivityThreshold()
            react_time = self.parameter_manager.getReactionTime()
            response_factor = self.parameter_manager.getResponseFactor()
            recovery_time = self.parameter_manager.getRecoveryTime()

            # Pack the data
```

```
            data_packet = struct.pack(data_format, 0x16, 0x55, mode, lrl, url, msr,
aa, va, apw, vpw, asens, vsens, arp, vrp, pvarp, act_thresh, react_time,
response_factor, recovery_time)
            print(f"Packet Length: {len(data_packet)}")
            ser.write(data_packet)

            # Wait for a response and read it
            time.sleep(0.5)
            response_data = ser.read(71)  ############## DATA LENGTH IS HERE
            ser.close()

            # Debug data type
            if len(response_data) != 71:
                QMessageBox.warning(self, "Error", "Invalid data length received from
pacemaker.")
                return
            # Unpack the response values
            modeV = struct.unpack('B', response_data[0:3])
            lrlV = struct.unpack('f', response_data[4:7])
            urlV = struct.unpack('f', response_data[8:11])
            msrV = struct.unpack('f', response_data[12:15])
            aaV = struct.unpack('f', response_data[16:19])
            vaV = struct.unpack('f', response_data[20:23])
            apwV = struct.unpack('f', response_data[24:27])
            vpwV = struct.unpack('f', response_data[28:31])
            asensV = struct.unpack('f', response_data[32:35])
            vsensV = struct.unpack('f', response_data[36:39])
            arpV = struct.unpack('H', response_data[40:43])
            vrpV = struct.unpack('H', response_data[44:47])
            pvarpV = struct.unpack('H', response_data[48:49])
            act_threshV = struct.unpack('f', response_data[50:53])
            react_timeV = struct.unpack('f', response_data[54:57])
            response_factorV = struct.unpack('f', response_data[58:61])
            recovery_timeV = struct.unpack('f', response_data[62:65])

            # Debugging recived data
            print(f"Data received:")
            print(f"Mode: {modeV}, Lower Rate Limit: {lrlV}, Upper Rate Limit:
{urlV}, Maximum Sensor Rate: {msrV}")
            print(f"Atrial Amplitude: {aaV}, Ventricular Amplitude: {vaV}, Atrial
Pulse Width: {apwV}, Ventricular Pulse Width: {vpwV}")
            print(f"Atrial Sensitivity: {asensV}, Ventricular Sensitivity: {vsensV},
ARP: {arpV}, VRP: {vrpV}, PVARP: {pvarpV}")
```

```
            print(f"Activity Threshold: {act_threshV}, Reaction Time: {react_timeV},
Response Factor: {response_factorV}, Recovery Time: {recovery_timeV}")
           # Validate the response data
            if (modeV == mode and lrlV == lrl and urlV == url and msrV == msr and
abs(aaV - aa) < 0.01 and
                    abs(vaV - va) < 0.01 and apwV == apw and vpwV == vpw and asensV
== asens and vsensV == vsens and
                    arpV == arp and vrpV == vrp and pvarpV == pvarp and
abs(act_threshV - act_thresh) < 0.01 and
                    react_timeV == react_time and response_factorV == response_factor
and recovery_timeV == recovery_time):
                QMessageBox.information(self, "Success", "Parameters set and stored
successfully.")
            else:
                QMessageBox.warning(self, "Error", "Some or all parameters did not
store properly.")
        except struct.error as e:
            QMessageBox.warning(self, "Error", f"Data Packing/Unpacking Error:
{str(e)}")
        except serial.SerialException as e:
            QMessageBox.warning(self, "Error", f"Serial Communication Error:
{str(e)}")
        finally:
            if ser and ser.is_open:
                ser.close()
    # Method to start the Egram plot
    def start_egram(self):
        if self.connected_port:
            # Pass the connected port to the EgramPlot instance
            self.egram_plot = EgramPlot(port=self.connected_port)
            self.egram_plot.show()
        else:
            QMessageBox.warning(self, "Error", "Device not connected. Please connect
to start Egram.")

    # Method to exit the application
    def exit_program(self):
        QApplication.quit()
```

3. ParameterManager.py

```
class ParameterManager:
    def __init__(self):
        # Common parameters
        self.__lrl = 60.0  # single (float)
        self.__url = 120  # single (float)
        self.__msr = 120  # single (float)
        self.__fixed_av_delay = 150  # single (float)
        self.__atrial_amplitude = 5.0  # single (float)
        self.__ventricular_amplitude = 5.0  # single (float)
        self.__atrial_pulse_width = 1.0  # single (float)
        self.__ventricular_pulse_width = 1.0  # single (float)
        self.__atrial_sensitivity = 0.1  # single (float)
        self.__ventricular_sensitivity = 0.1  # single (float)
        self.__arp = 250  # single (float)
        self.__vrp = 320  # single (float)
        self.__pvarp = 250  # uint16
        self.__hysteresis = 0  # uint8
        self.__rate_smoothing = 0  # uint8
        self.__activity_threshold = 1.6  # single (float)
        self.__reaction_time = 30.0  # single (float)
        self.__response_factor = 8  # single (float)
        self.__recovery_time = 300.0  # single (float), sec = 5 min

    # Getters for common parameters
    def getLowerRateLimit(self):
        return self.__lrl

    def getUpperRateLimit(self):
        return self.__url

    def getMaximumSensorRate(self):
        return self.__msr

    def getFixedAVDelay(self):
        return self.__fixed_av_delay

    def getAtrialAmplitude(self):
        return self.__atrial_amplitude if self.__atrial_amplitude != 0 else 0.0

    def getVentricularAmplitude(self):
        return self.__ventricular_amplitude if self.__ventricular_amplitude != 0
else 0.0
```

```
def getAtrialPulseWidth(self):
    return self.__atrial_pulse_width

def getVentricularPulseWidth(self):
    return self.__ventricular_pulse_width

def getAtrialSensitivity(self):
    return self.__atrial_sensitivity

def getVentricularSensitivity(self):
    return self.__ventricular_sensitivity

def getARP(self):
    return self.__arp

def getVRP(self):
    return self.__vrp

def getPVARP(self):
    return self.__pvarp

def getHysteresis(self):
    return self.__hysteresis if self.__hysteresis != 0 else 0

def getRateSmoothing(self):
    return self.__rate_smoothing

def getActivityThreshold(self):
    return self.__activity_threshold

def getReactionTime(self):
    return self.__reaction_time

def getResponseFactor(self):
    return self.__response_factor

def getRecoveryTime(self):
    return self.__recovery_time / 60.0

# Setters for common parameters
def setLowerRateLimit(self, val):
    if self.__is_num(val):
        num = float(val)
```

```
            if 30.0 <= num <= 175.0:
                self.__lrl = num
            else:
                raise IndexError
        else:
            raise TypeError

    def setUpperRateLimit(self, val):
        if self.__is_num(val):
            num = float(val)
            if 50.0 <= num <= 175.0 and num >= self.__lrl:
                self.__url = num
            else:
                raise IndexError
        else:
            raise TypeError

    def setMaximumSensorRate(self, val):
        if self.__is_num(val):
            num = float(val)
            if 50.0 <= num <= 175.0 and num >= self.__lrl:
                self.__msr = num
            else:
                raise IndexError
        else:
            raise TypeError

    def setFixedAVDelay(self, val):
        if self.__is_num(val):
            if 70.0 <= float(val) <= 300.0:
                self.__fixed_av_delay = float(val)
            else:
                raise IndexError
        else:
            raise TypeError

    def setAtrialAmplitude(self, val):
        if str(val).casefold() == 'off'.casefold():
            self.__atrial_amplitude = 0.0
            return
        if self.__is_num(val):
            num = round(float(val), 1)
            if 0.1 <= num <= 5.0:
                self.__atrial_amplitude = num
```

```
            else:
                raise IndexError
        else:
            raise TypeError

    def setVentricularAmplitude(self, val):
        if str(val).casefold() == 'off'.casefold():
            self.__ventricular_amplitude = 0.0
            return
        if self.__is_num(val):
            num = round(float(val), 1)
            if 0.1 <= num <= 5.0:
                self.__ventricular_amplitude = num
            else:
                raise IndexError
        else:
            raise TypeError

    def setAtrialPulseWidth(self, val):
        if self.__is_num(val):
            if 1.0 <= float(val) <= 30.0:
                self.__atrial_pulse_width = float(val)
            else:
                raise IndexError
        else:
            raise TypeError

    def setVentricularPulseWidth(self, val):
        if self.__is_num(val):
            if 1.0 <= float(val) <= 30.0:
                self.__ventricular_pulse_width = float(val)
            else:
                raise IndexError
        else:
            raise TypeError

    def setAtrialSensitivity(self, val):
        if self.__is_num(val):
            if 0.0 <= float(val) <= 5.0:
                self.__atrial_sensitivity = round(float(val), 1)
            else:
                raise IndexError
        else:
            raise TypeError
```

```
def setVentricularSensitivity(self, val):
    if self.__is_num(val):
        if 0.0 <= float(val) <= 5.0:
            self.__ventricular_sensitivity = round(float(val), 1)
        else:
            raise IndexError
    else:
        raise TypeError

def setARP(self, val):
    if self.__is_num(val):
        if 150.0 <= float(val) <= 500.0:
            self.__arp = float(val)
        else:
            raise IndexError
    else:
        raise TypeError

def setVRP(self, val):
    if self.__is_num(val):
        if 150.0 <= float(val) <= 500.0:
            self.__vrp = float(val)
        else:
            raise IndexError
    else:
        raise TypeError

def setPVARP(self, val):
    if self.__is_num(val):
        if 150 <= int(val) <= 500:
            self.__pvarp = int(val)
        else:
            raise IndexError
    else:
        raise TypeError

def setHysteresis(self, val):
    if str(val).casefold() == 'off'.casefold():
        self.__hysteresis = 0
        return
    if self.__is_num(val):
        num = int(float(val))
        if 0 <= num <= 175:
```

```
            self.__hysteresis = num
        else:
            raise IndexError
    else:
        raise TypeError

def setRateSmoothing(self, val):
    if self.__is_num(val):
        num = int(val)
        if 0 <= num <= 25:
            self.__rate_smoothing = num
        else:
            raise IndexError
    else:
        raise TypeError

def setActivityThreshold(self, val):
    if self.__is_num(val):
        self.__activity_threshold = float(val)
    else:
        raise TypeError

def setReactionTime(self, val):
    if self.__is_num(val):
        num = float(val)
        if 10.0 <= num <= 50.0:
            self.__reaction_time = num
        else:
            raise IndexError
    else:
        raise TypeError

def setResponseFactor(self, val):
    if self.__is_num(val):
        num = float(val)
        if 1.0 <= num <= 16.0:
            self.__response_factor = num
        else:
            raise IndexError
    else:
        raise TypeError

def setRecoveryTime(self, val):
    if self.__is_num(val):
```

```
            num = float(val)
            if 1.0 <= num <= 16.0:
                self.__recovery_time = num * 60
            else:
                raise IndexError
        else:
            raise TypeError

    def __is_num(self, s):
        try:
            float(s)
        except ValueError:
            return False
        else:
            return True
```

4. egram_plot.py

```
import sys
import numpy as np
import serial
import struct
import time
from PyQt6.QtCore import QTimer
from PyQt6.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QPushButton,
QWidget, QMessageBox
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure

class EgramPlot(QMainWindow):
    def __init__(self, port=None):
        super().__init__()
        self.port = port  # Connected serial port
        self.aData = np.array([])  # Data for atrium signals
        self.vData = np.array([])  # Data for ventricle signals
        self.tData = np.array([])  # Time data
        self.cont = True  # Continue plotting
        self.vS = True  # Visibility of ventricle signals
        self.aS = True  # Visibility of atrium signals
        self.start_time = time.time()
```

```
# Set up the window
self.setWindowTitle('Egram')
self.setGeometry(100, 100, 800, 600)

# Set up the central widget and layout
self.central_widget = QWidget(self)
self.setCentralWidget(self.central_widget)
self.layout = QVBoxLayout(self.central_widget)

# Set up the plot
self.fig = Figure(figsize=(7, 7), dpi=100)
self.plotA = self.fig.add_subplot(211)
self.plotV = self.fig.add_subplot(212)
self.fig.subplots_adjust(hspace=0.8)
self.plotA.set_title('Atrium Signals', fontsize=12)
self.plotV.set_title('Ventricle Signals', fontsize=12)
self.plotA.set_xlabel("Time (sec)", fontsize=10)
self.plotA.set_ylabel("Voltage (V)", fontsize=10)
self.plotV.set_xlabel("Time (sec)", fontsize=10)
self.plotV.set_ylabel("Voltage (V)", fontsize=10)
self.plotA.set_ylim(-6, 6)
self.plotA.set_xlim(0, 16)
self.linesA = self.plotA.plot([], [])[0]
self.plotV.set_ylim(-6, 6)
self.plotV.set_xlim(0, 16)
self.linesV = self.plotV.plot([], [])[0]

# Add plot layout
self.canvas = FigureCanvas(self.fig)
self.layout.addWidget(self.canvas)

# Button for start atrium graph
self.a_button = QPushButton('Atrium', self)
self.a_button.clicked.connect(self.toggle_atrium_plot)
self.layout.addWidget(self.a_button)
# Button for start ventricle graph
self.v_button = QPushButton('Ventricle', self)
self.v_button.clicked.connect(self.toggle_ventricle_plot)
self.layout.addWidget(self.v_button)
# Button for start and stop data transmitting
self.start_stop_button = QPushButton('Start/Stop', self)
self.start_stop_button.clicked.connect(self.toggle_plot)
self.layout.addWidget(self.start_stop_button)
```

```
        # Start plotting
        self.plot()

    def plot(self):
        if self.cont:
            try:
                if self.port:
                    # Recived data from pacemaker
                    with serial.Serial(port=self.port, baudrate=115200, timeout=1)
as ser:
                        ser.write(struct.pack('<2B', 0x16, 0x22))  # Request data
                        serial_data = ser.read(71)
                        if len(serial_data) != 71:
                            raise serial.SerialException("Incomplete data
received.")

                        # Process received data
                        a_signal = -6.6 * (struct.unpack('<f', serial_data[4:8])[0] -
0.5)
                        v_signal = -6.6 * (struct.unpack('<f', serial_data[8:12])[0] -
0.5)
                else:
                    # Generate random data if no pacemaker is connected
                    a_signal = np.random.uniform(-5, 5)
                    v_signal = np.random.uniform(-5, 5)

                # Update data arrays
                if len(self.aData) < 300:
                    self.aData = np.append(self.aData, a_signal)
                    self.vData = np.append(self.vData, v_signal)
                    self.tData = np.append(self.tData, time.time() -
self.start_time)
                else:
                    self.aData[:-1] = self.aData[1:]
                    self.vData[:-1] = self.vData[1:]
                    self.tData[:-1] = self.tData[1:]
                    self.aData[-1] = a_signal
                    self.vData[-1] = v_signal
                    self.tData[-1] = time.time() - self.start_time
                    self.plotA.set_xlim(self.tData[0], self.tData[-1])
                    self.plotV.set_xlim(self.tData[0], self.tData[-1])

                # Update plot lines
                self.linesA.set_data(self.tData, self.aData)
```

```
                self.linesV.set_data(self.tData, self.vData)
                self.canvas.draw()

            except (serial.SerialException, struct.error) as e:
                QMessageBox.warning(self, "Error", f"Error: {str(e)}. Switching to
random data.")
                self.port = None  # Fallback to random data

        QTimer.singleShot(50, self.plot)  # Refresh every 50ms

    def toggle_atrium_plot(self):
        self.aS = not self.aS
        self.plotA.set_visible(self.aS)
        self.canvas.draw()

    def toggle_ventricle_plot(self):
        self.vS = not self.vS
        self.plotV.set_visible(self.vS)
        self.canvas.draw()

    def toggle_plot(self):
        self.cont = not self.cont
        if self.cont:
            self.start_time = time.time() - self.tData[-1] if len(self.tData) > 0
else time.time()

# Only for test the egram functionalities
if __name__ == '__main__':
    app = QApplication(sys.argv)
    egram = EgramPlot(port="COM6")  # Example port for testing
    egram.show()
    sys.exit(app.exec())
```

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. load_user_parameters(self)

- Parameters: None

C. save_user_parameters(self)

- Parameters: None

D. get_relevant_parameters_for_mode(self, mode)

- Parameters: mode (string) – represents the selected pacing mode

E. get_mode_value(self, mode)

- Parameters: mode (string) – represents the selected pacing mode

F. apply_parameters(self)

- Parameters: None

G. update_parameters(self)

- Parameters: None

H. set_field_visibility(self, visibility_dict)

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

I. check_connection_status(self)

- Parameters: None

J. send_and_validate_parameters(self)

- Parameters: None

K. start_egram

- Parameters: None

M. exit_program(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. load_user_parameters: reads the user data file (users.dat) and loads stored parameters. If the file doesn't exist, it initializes an empty dictionary.

C. save_user_parameters: saves the current user's parameters into the users.dat file and overwrites previously data for that user

D. get_relevant_parameters_for_mode: returns list of parameter names for the selected pacing mode

E. get_mode_value: translates pacing mode string into an integer value to be used when communicating between devices

F. apply_parameters: checks validity of the parameter values entered by the user in the GUI. If valid, sends the inputs to the connected pacemaker and updates the user's parameters. If an error occurs, displays warning message.

G. Update_parameters: when pacing mode changed, determines which parameters should be visible based on the selected mode and updates the GUI accordingly.

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

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

J. send_and_validate_parameters: sends data over a serial connection to the pacemaker device, waits for a response and validates whether the parameters were correctly received. If they match, success message displayed; otherwise, an error message displayed.

K. start_egram: if device connected, an Egram plot is displayed, reflecting the pacemaker data. If device not connected, error message shown

L. sxit_program: hides the application window, basically exiting it.

Global Variables (State Variables):

A. self.parameter_manager: responsible for managing and storing the pacing parameters for the current user and allows for retrieving and updating the values for various pacing parameters

B. self.username: username of current user logged in.

C. self.users_file: the file path where user parameters are stored. Used for loading and saving user data to and from the file.

D. self.users_parameters: dictionary that holds parameters for each users. The keys are username, the values are the objects that store the user's settings.

E. self.connected_port: stores name of serial port currently connected to pacemaker device.

F. self.fields: contains the widgets for various parameters, allowing for

dynamic access to input fields throughout the class.

G. self.field_containers: dictionary that contains the pairs of labels and field widgets for each parameter. Allow the class to manage the visibility and layout of the parameter input fields.

H. self.pacing_mode_combo: allows the user to select a pacing mode and allows visible parameters to be updated based on the selected pacing mode

I. self.status_msg: shows connection status. Updated when connection checked

J. self.exit_button: button that allows user to exit application. Triggers exit_program method when clicked.

K. self.apply_button: button that applies the selected parameters to the ParameterManager and tries to send them to the pacemaker.

L. self.egram_button: button that starts Egram plot when clicked, if device is connected.

M. self.refresh_button: a button that checks the connection status when clicked

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

Internal Behaviour of Each Function:

A. __init__:

- creates an instance of ParameterManager (self.parameter_manager)
- sets self.username and loads user parameters from a file (self.load_user_parameters)
- Initializes a dictionary (self.fields) and populates parameter fields (QLineEdit and QComboBox) with values from parameter_manager based on the current user
- Initializes window title (“application window”) and dimensions (800x600 pixels)
- Initializes username label (self.username_label) - a QLabel displaying the currently user’s name.
- 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. load_user_parameters:

- Reads users.dat file using pickle and assigns the loaded data to self.user_parameters. If file does not exist, initializes it as an empty dictionary.

C. save_user_paramters:

- Self.parameter_manager saved into self.user_parameters under the current user's username
- The self.user_parameters dictionary is serialized back to the users.dat file, updating the stored parameter values.

D. get_relevant_parameters_for_mode:

- based on selected pacing mode, looks up relevant parameters from relevant_params_dict and returns the required information.
- Determines which parameters should be visible or can be edited based on selected pacing mode

E. get_mode_value:

- Based on selected pacing mode (mode), returns an integer that corresponds to the selected mode to communicate with pacemaker device later on

F. apply_parameters:

- Retrieves selecting pacing mode and its parameters
- Loops over all form fields and applies values to the parameter_manager only for fields relevant to current mode
- Validates correct value types and ranges
- Displays error message for invalid input using QMessageBox
- If parameters applied successfully, the parameter_manager is updated and the user parameters are saved

G. update_parameters:

- When pacing mode changed, calls get_relevant_parameters_for_mode to get the fields relevant to the mode
- Visibility of form fields (QLineEdit, QComboBox, etc.) updated to show only those that are relevant to selected pacing mode

H. set_field_visibility:

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

I. check_connection_status:

- Iterates over ports and attempts to establish a serial connection
- If connection successful, port saved in self.connected_port and success message is shown
- Updates status message (self.status_msg). if no device connected, “Device not connected” is shown.

J. send_and_validate_parameters:

- Data is send to pacemaker, received data is compared with the parameters to ensure they match
- If matched, success message shown. If discrepancies found, error message shown.
- This method does not directly alter the internal parameters but ensures the device accepts and correctly stores the parameters

K. start_egram:

- If device connected, an EgramPlot instance created with the connected serial port shown as a parameter.
- The state of the self.connected_port is checked, ensuring egram can only be viewed when a device is connected

L. exit_program:

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

3. egram_plot.py

Purpose: Implements a GUI to visualize and monitor atrium and ventricle signals coming from a pacemaker connected via a serial port. The signals are plotted in real-time using separate subplots for atrium and ventricle data. Can toggle the visibility of the plots as well as start and stop the data transmission.

Public Functions:

A. __init__(self, port=None)

- Parameters: port(optional) a string representing the serial port for pacemaker data transmission

B. Plot(self)

- Parameters: None

C. toggle_atrium_plot(self)
    - Parameters: None

D. toggle_ventricle_plot(self)
    - Parameters: None

E. toggle_plot(self)
    - Parameters: None

Black-Box Behaviour of Each Function:

A. __init__: initializes GUI window and plot, sets up data arrays, configures buttons, sets up serial port communication if a port is provided, or defaults to random data if not

B. plot: continuously plots atrium and ventricle signals. If a valid serial port is connected, it obtains and processes the signals from the pacemaker; otherwise it generates random signal data. Updates the plot with new data and refreshes canvas. If an error occurs, it switches to random data and notifies user.

C. toggle_atrium_plot: toggles the visibility of the atrium signal plot.

D. Toggle_ventricle_plot: toggles the visibility of the ventricle signal plot

E. Toggle_plot: starts or stops the real-time plotting of data. If stopped, it will no longer obtain data and update the plots. If started again, it will continue the data update from where it left off.

Global Variable:

A. Self.port (type: str or None): represents the serial port to which the pacemaker is connected. If no port, defaults to None, and random data used instead

B. Self.aData: stores atrium signal data points

C. Self.vData: stores ventricle signal data points

D. Self.tData: stores time values corresponding to atrium and ventricle signals

E. Self.cont: controls whether data transmission and plotting should continue or stop

F. Self.vS: controls visibility of the ventricle signal plot

G. Self.aS: controls visibility of the atrium signal plot

H. Self.start_time: stores start time of the plotting process

I. Self.plotA: the axes object for the atrium plot

J. Self.plotV: the axes object for the ventricle plot

K. Self.linesA: the plot line object for atrium signal data

L. Self.linesV: the plot line object for ventricle signal data

M. Self.canvas: the canvas object used to display the plot

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

Internal Behaviour of Each Function:

A. __init__:

- GUI initialized
- Initializes aData, vData, and tData as empty arrays to store the signal data
- Sets cont. flag to True to enable continuous plotting
- Two buttons set up to toggle visibility of plots
- Start_stop_button created to toggle start/stop of data transmission
- If serial port provided, prepares to receive data from pacemaker
- Starts first data plot with plot()

B. Plot:

- Runs in a loop, repeatedly getting and updating data every 50ms
- If self.cont is True, gets pacemaker data via serial communication or simulates random values for the signals
- The new signal appended to the aData, vData, and tData arrays. If these arrays exceed 300 elements, the older elements are discarded and replaced with new data
- Plot axes are updated with new data and the canvas is redrawn
- If there is an error, switches to random data and updates UI with an error message

C. Toggle_atrium_plot:

- Toggles visibility of atrium signal plot based on current state of self.aS.

if True, plot is made visible; otherwise, it is hidden

D. Toggle_ventricle_plot:

- Toggles the visibility of the ventricle signal plot based on current state of self.vS. if True, plot is made visible; otherwise, it is hidden.

E. Toggle_plot:

- Toggles the cont. flag which determines whether data transmission and plotting should continue. If false, data transmission stops and if True, it resumes, and start_time is adjusted to account for the elapsed time since the last data point.

3. Parameter_Manager.py

Purpose: responsible for maintaining a collection of parameters related to the functionality of the pacemaker. Provides methods for both retrieving and setting the values of various parameters that control how the system behaves. It also ensures parameters are set within predefined valid ranges and types

Public Functions:

A. getLowerRateLimit(self)

B. getUpperRateLimit(self)

C. getMaximumSensorRate(self)

D. getFixedAVDelay(self)

E. getAtrialAmplitude(self)

F. getVentricularAmplitude(self)

G. getAtrialPulseWidth(self)

H. getVentricularPulseWidth(self)

I. getAtrialSensitivity(self)

J. getVentricularSensitivity(self)

K. getARP(self)

L. getVRP(self)

M. getPVARP(self)

N. getHysteresis(self)

O. getRateSmoothing(self)

P. getActivityThreshold(self)

Q. getReactionTime(self)

R. getResponseFactor(self)

S. getRecoveryTime(self)

the above are all getter functions. Each getter function has a corresponding setter function (e.g setARP, setVRP, etc.). None of these functions has a parameter.

Black-Box Behaviour of Each Function:

Each getter function simply returns the current value of the respective parameter (e.g getLowerRateLimit returns current Lower Rate Limit value as a float)

Each setter function takes a value val, checks if the value is of the correct type, and validates that the value falls within a specific range. If the value is valid, the parameter is updated. If invalid, an exception is raised

Global Variables (State Variable): does not use global variables. All state information stores in instance variables within the class (e.g. __lrl: lower rate limit (float))

Private Functions:

A. ___is_num(self,s): private helper function to check if a string s can be converted to a number(float). Returns True if s is a valid number, otherwise returns false.

Internal Behaviour of Functions:

A. Getter functions:

- Each getter function retrieves the value of the corresponding state variable. For example, getLowerRateLimit retrieves the value of __lrl

B. Setter Functions:

- Validates the input val (checks if val is a number and whether the value falls within parameter range)
- If value valid, updates corresponding internal state variable. If not valid, displays exception.

C. __is_num:

- Attempts to convert a string to a float using float(s). if successful, returns True, if not, returns False, meaning the string is not a valid number.