Pacemaker and DCM Development Documentation

SFWRENG/MECHTRON 3K04 Assignment 1 and 2

Group 33

November 29, 2024

Ahmet Asan, asana@mcmaster.ca

Xiaojun Shen, shenx46@mcmaster.ca

(Brad) Ziyi Wang, wangz838@mcmaster.ca

Xikai Xu, xu503@mcmaster.ca

Kevin Yu, yuk51@mcmaster.ca

Tianyao Zhang, zhant88@mcmaster.ca

Table of Contents

- **Part 3: Current Progress and Modified Features**
    - 3.1 Major Cyclic States of Pacing Process
    - 3.2 Modified Features from Assignment
    - 3.3 Changes in Requirements and Design Decisions
    - 3.4 Bonus Mode: DDDR
- **Part 4: Safety Assurance**
    - 4.1 Approach
    - 4.2 Standards
    - 4.3 Testing
- **Part 5: Testing and Results**
    - 5.1 AOO Testing
    - 5.2 VOO Testing
    - 5.3 AAI Testing
    - 5.4 VVI Testing
    - 5.5 AOOR Testing
    - 5.6 VOOR Testing
    - 5.7 AAIR Testing
    - 5.8 VVIR Testing
- **Part 6: Simulink Model Screenshots**
- **Part 7: DCM Design Documentation**

Part 1: Simulink Design Documentation

1. Requirements

This section defines the functional and operational requirements needed to ensure the proper performance of the pacemaker system. Each requirement is linked to specific design and functional elements to maintain traceability.

1.1 Required inputs and corresponding Regulation

Lower Rate Limit (LRL) Range**:** 30 to 175 BPM

Description**:** Minimum heart rate at which the pacemaker ensures pacing.

Upper Rate Limit (URL) Range**:** 50 to 200 BPM

Description**:** Maximum rate at which the pacemaker will pace.

Atrial Pulse Amplitude Range**:** 0.5 to 7.0 V (3.5V)

Description**:** Voltage amplitude of the pacing pulse to the atrium.

Ventricular Pulse Amplitude Range**:** 0.5 to 7.0 V (3.5V)

Description**:** Voltage amplitude for ventricular pacing.

Atrial Pulse Width Range**:** 0.1 to 1.9 ms (0.4ms)

Description**:** Duration of the pacing pulse delivered to the atrium.

Ventricular Pulse Width Range**:** 0.1 to 1.9 ms (0.4ms)

Description**:** Duration of the pacing pulse delivered to the ventricle.

Atrial Refractory Period (ARP) Range**:** 150 to 500 ms (320 ms)

Description: Period during which atrial signals are ignored to avoid double counting.

Ventricular Refractory Period (VRP) Range**:** 150 to 500 ms (320 ms)

Description**:** Period during which ventricular signals are ignored to prevent false triggering.

Pacing Reference PWM: No designated ranges, according to PWM calculation and preferred 3.5V amplitude, 3.5V/5.0V = 70% is used.

Description**:** The Reference PWM to establish a natural heart activity sensing threshold.

Mode Selection: No designated ranges, implement in current stage of design to choose which mode to perform.

Description**:** Value 0 = AOO mode, Value 1 = VOO mode, Value 2 = AAI mode, Value 3 = VVI mode, Value 4 = AOOR mode, Value 5 = VOOR mode, Value 6= AAIR mode, Value 7 = VVIR mode.

Max Sensor Rate (MSR): Range 50 – 175 BPM, Default: 120 BPM

Description: Max sensor rate, Adaptive rate cannot exceed this heart rate.

Reaction_time: Range 5 – 30 s, Default: 10 seconds

Description: The time the pacemaker takes to ramp up to the adaptive rate pace from the regular pace.

Response_factor: Range 1 – 16, Default: 8

Description: The increase in adaptive heart rate, given a certain increase in activity.

Recovery Time: Range 10 – 60 s, Default: 30 s

Description: The time it takes for the pacemaker to return to standard pacing after the motion has concluded.

Parameter	Programmable Values	Increment	Nominal	Tolerance	MODE
Lower Rate Limit	30-50 ppm, 50-90 ppm, 90-175 ppm	5 ppm, 1 ppm	60 ppm	±8 ms	ALL
Upper Rate Limit	50-175 ppm	5 ppm	120 ppm	±8 ms	ALL
Maximum Sensor Rate	50-175 ppm	5 ppm	120 ppm	±4 ms	AOOR, VOOR, AAIR, VVIR
Fixed AV Delay	70-300 ms	10 ms	150 ms	±8 ms	
Dynamic AV Delay	Off, On	—	Off	—	
Minimum Dynamic AV Delay	30-100 ms	10 ms	50 ms	±1 ms	
Sensed AV Delay Offset	Off, -10 to -100 ms	10 ms	Off	±1 ms	
A or V Pulse Amplitude Regulated	Off, 0.5-3.2V, 3.5-7.0V	0.1V, 0.5V	3.5V	±12%	ALL
A or V Pulse Amplitude Unregulated	Off, 1.25, 2.5, 3.75, 5.0V	—	3.75V	—	ALL
A or V Pulse Width	0.05 ms, 0.1-1.9 ms	0.1 ms	0.4 ms	0.2 ms	ALL
A or V Sensitivity	0.25, 0.5, 0.75, 1.0-10 mV	0.5 mV	A-0.75 mV, V-2.5 mV	±20%	AAI, VVI, AAIR, VVIR
Ventricular Refractory Period (VRP)	150-500 ms	10 ms	320 ms	±8 ms	VVI, VVIR
Atrial Refractory Period (ARP)	150-500 ms	10 ms	250 ms	±8 ms	AAI, AAIR
PVARP	150-500 ms	10 ms	250 ms	±8 ms	AAI, AAIR
PVARP Extension	Off, 50-400 ms	50 ms	Off	±8 ms	
Hysteresis Rate Limit	Off or same choices as LRL	—	Off	±8 ms	AAI, VVI, AAIR, VVIR
Rate Smoothing	Off, 3, 6, 9, 12, 15, 18, 21, 25%	—	Off	±1%	AAI, VVI, AAIR, VVIR
ATR Mode	On, Off	—	Off	—	
ATR Duration	10 cardiac cycles, 20-80 cc, 100-2000 cc	20 cc, 100 cc	20 cc, 100 cc	±1 cc	
ATR Fallback Time	1-5 min	1 min	1 min	±30 sec	
Ventricular Blanking	30-60 ms	10 ms	40 ms	—	
Activity Threshold	V-Low, Low, Med-Low, Med, Med-High, High, V-High	—	Med	—	AOOR, VOOR, AAIR, VVIR
Reaction Time	10-50 sec	10 sec	30 sec	±3 sec	AOOR, VOOR, AAIR, VVIR
Response Factor	1-16	1	8	—	AOOR, VOOR, AAIR, VVIR
Recovery Time	2-16 min	1 min	5 min	±30 sec	AOOR, VOOR, AAIR, VVIR

Table of Programmable Variables and relationship with Modes

1.2 Mode-based Pacing and Sensing

AOO Mode: The pacemaker must continuously stimulate the atrium, regardless of natural activity, at a predefined interval to maintain atrial contraction.

VOO Mode: The pacemaker must provide regular pulses to the ventricle to ensure steady ventricular contraction, even if natural beats occur.

AAI Mode: In this mode, the pacemaker only delivers atrial pulses if it does not detect any natural atrial activity within a predefined interval, conserving energy and ensuring synchronization.

VVI Mode: The system only stimulates the ventricle when no natural ventricular activity is detected, avoiding unnecessary pacing pulses, and maintaining efficiency.

Rate Adaptive Modes: AOOR, VOOR, AAIR, VVIR are almost identical to the respective modes without rate adaptive characteristic, since the only difference is that they uses the adapted rate coming from the rate adaptive module instead of LRL.

Rate Adaptive sensing: Rate-adaptive sensing refers to the increase in pacing activity based on the movement of the patient. When the patient starts moving or running, the pacemaker responds by increasing the patient's heart rate. After detecting that the patient has stopped moving, the pacemaker gradually decreases the patient's heart rate.

This is an additional layer applied to AOO, VOO, AAI, and VVI modes. When rate-adaptive sensing is included, the modes are referred to as AOOR, VOOR, AAIR, and VVIR.

1.3 Pin Control Operations

Each pin must serve a precise function for pacing, sensing, or controlling the impedance circuitry:

D0 (ATR CMP DETECT): Outputs ON (HIGH) when atrial signal voltage is higher than the threshold.

D1 (VENT CMP DETECT): Same functionality as D0 but for ventricular signals.

D2 (PACE CHARGE CTRL): Controls charging of the primary capacitor. It is crucial to ensure this pin is never set to HIGH when ATR or VENT PACE CTRL is active, as it risks delivering inappropriate pacing.

D3 (VENT CMP REF PWM): PWM input to establish a ventricular sensing threshold.

D4 (Z ATR CTRL): Controls impedance measurement in the atrial electrode.

D5 (PACING_REF_PWM): Used to charge the primary capacitor (C22) of the pacing circuit.

D6 (ATR_CMP_REF_PWM): Same functionality as in VENT_CMP_REF PWM but for the atrial action potential.

D7 (Z_VENT_CTRL): This control allows the impedance circuit to be connected to the ring electrode of the ventricle. Its use is identical to Z_ATR_CTRL but for the ventricle.

D8 (ATR PACE CTRL) and **D9 (VENT PACE CTRL)**: Used to control pacing signals to the atrium and ventricle respectively.

D9 (VENT_PACE_CTRL): Same functionality as in ATR PACE CTRL but for the ventricle.

D10 (PACE GND CTRL): Activates grounding to complete the circuit during pacing operations.

D11 (ATR_GND_CTRL): Used to connect the ATR RING OUT to GND. This functionality is used when discharging the blocking capacitor through the atrium to allow no charge buildup.

D12 (VENT_GND_CTRL): Same functionality as in ATR RING OUT but for the ventricle.

D13 (FRONTEND_CTRL): Used to activate the sensing circuitry.

If ON (HIGH) →Sensing circuitry will output heart signal.

If OFF (LOW) →Sensing circuitry is disconnected from patient and will output nothing.

6-AXES Sensor (Accelerometer): Used to detect motion in the pacemaker, outputs a vector.

1.4 Capacitor Management

The C22 capacitor must be charged via the PWM input to prepare for pacing pulses.
After each pulse, C21 (blocking capacitor) must be discharged to prevent charge buildup, using the corresponding GND_CTRL pins.

1.5 Mode Selection

The pacemaker must allow seamless switching between modes via the Mode input signal:

- Mode = 0 → AOO
- Mode = 1 → VOO
- Mode = 2 → AAI
- Mode = 3 → VVI
- Mode = 4 → AOOR
- Mode = 5 → VOOR
- Mode = 6 → AAIR
- Mode = 7 → VVIR
- Mode = 8 → DDDR (FOR BONUS)

1.6 Refractory Periods (VRP & ARP) and Rate Adaptive Calculations

After each paced or detected event, the pacemaker must initiate a Ventricular Refractory Period (VRP) or Atrial Refractory Period (ARP). These periods, ranging from 150 to 500 ms, ensure that false signals and after-potentials do not cause unintended pacing.

To determine the effect of the motion of the patient on heart rate, Vectors from the 6-Axes Accelerometer must be taken and a series of calculations must be applied. This should output a modified heart rate based on the motion.

1.7 PWM Control and Amplitude Management

PWM signals are required to regulate pacing amplitudes and sensing thresholds. For a 5V system, a 70% duty cycle is used to achieve a 3.5V amplitude.

1.8 Serial Communication

The DCM team must be able to send inputs such as the required mode, the pacing rate and other information, the pacemaker must then be able to act on this information and return an EGRAM via serial communication.

1.9 Anticipated Requirements for Future Development

To maintain the relevance and efficacy of the pacemaker software, it is essential to plan for future enhancements and the integration of additional features. A critical area of focus is the expansion of the pacemaker's operational modes and states, for example, the possible implementation of the DDDR mode. To ensure reliability in diverse environments, the software should also incorporate features for remote monitoring, enabling healthcare professionals to track and adjust pacemaker settings in real time. Integrating these functionalities will guarantee that the pacemaker remains at the forefront of patient care and continues to offer the highest standards of performance and safety.

On the other hand, future enhancements to the DCM system will focus on improving UX and security. The interface should be resizable on different devices with arbitrary screen sizes, with

visual alerts for low and full battery statuses. A two-factor authentication system will be added for secure login, and users will be able to request password recovery via email. The DCM will have a new feature to verify that the selected communication port is connected to a valid pacemaker device. The electrogram graph will be interactive, allowing resizing, unit adjustments, and color scheme customization. Admin access will be secured by requiring re-entry of the password when saving changes to settings, especially pacing modes. Additionally, real-time bradycardia reports will be available for healthcare providers, reflecting current electrogram data. These updates will ensure the DCM system is both user-friendly and secure.

2. Design Decisions

This section outlines the key design decisions that guided the development and implementation of the pacemaker.

2.1 Mode-based Design Approach

Each operating mode (AOO, VOO, AAI, VVI) is implemented as an independent state machine. This modular design ensures clear separation of responsibilities and makes debugging easier.

2.2 Refractory Period Handling

The VRP begins immediately after a pacing pulse is initiated or a natural beat is detected. This decision ensures no double counting occurs and that the heart is not overstimulated during recovery periods.

2.3 Capacitor Management

The C22 and C21 capacitors are combined in the current design to streamline state transitions and reduce complexity. However, this approach is under review for future iterations to provide more granular control.

2.4 Adaptive Rate

The adaptive rate subsystem takes several pre-defined values and combines them with data from the accelerometer to produce an adaptive rate which will be used as the pacing rate. This subsystem considers several key pre-defined values:
Activity Threshold: The activity level required before the modified adaptive pace rate replaces the previous pace rate, by default this is when the scalar value of the accelerometer passes 1.1 g.

2.4.1 Rate Adaptive Logic Stateflow Description

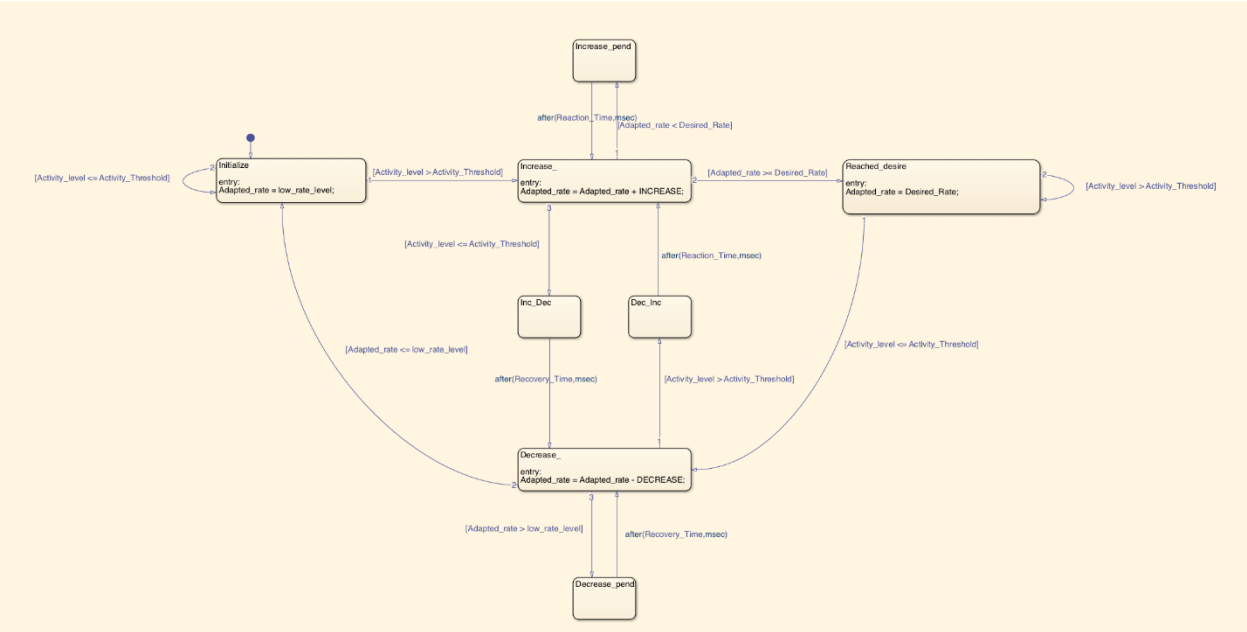
The rate adaptive module should output a adapted rate that will change in time according to the move of the pacemaker, and the adapted rate will be introduced into the main stateflow and work as the BPM for all rate adaptive modes.

In the main stateflow of Rare Adaptive module, the increase or decrease of adapted rate will depend on the comparison between the current activity level and activity threshold. When the activity level is bigger than threshold, the adapted rate will start to increase by an amount obtained by calculation. This amount depends on a list of parameters such as, LRL, MSR, Response Factor and Reaction time. The adapted rate will be no lower than LRL and no higher

than MSR (Max sensor rate). A high response factor and a low reaction time can make Adapted rate increase at a higher speed, until it reaches MSR.

When the activity level is smaller than threshold, the adapted rate will start to decrease by an amount obtained by calculation. This amount depends on a list of parameters LRL, MSR, Response Factor and Response time. A high response factor and a low response time can make Adapted rate decrease at a higher speed, until it reaches LRL.

The adapted rate is updated in real time and transferred to the main state flow.

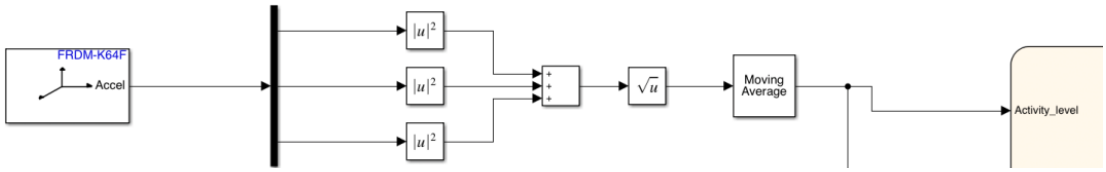


Rate Adaptive Module

2.4.2 Activity level calculation

The data received from the accelerometer is in vector format the conversion was done by taking each component and turning it into a scalar component, this scalar value was then used by the rest of the adaptive rate module to determine things like activity level and adapted rate (The modified heart rate because of the motion). As the data from the accelerometer is often volatile in

nature, a moving average block from the DSP toolbox in MATLAB was used to smooth out the activity level output of the conversion. Below is the process used to generate the scalar value.



2.5 Serial Communication

The serial communication was implemented in Simulink through a UART receiver, wherein the values, in binary, were converted to usable inputs on the Simulink side. After the processing was completed, appropriate outputs were sent back through the UART module.

2.5.1 Serial Communication Subsystem and architecture

The serial communication subsystem has two main modes of operation, with and without a UART signal. Without the UART signal the subsystem operates in default mode, with all the inputs being set to the default. The subsystem as mentioned above will also duplicate the parameter data to DCM to generate an ECG diagram when connected.

For receiving UART communication data, we use a part named UART receive in Simulink. It has two outputs, one is Rx and the other is Status. The Status output tells whether there is information and data being received, and Rx is the data itself received.

For sending UART communication data, we use a part named UART send in Simulink. It can send data in terms of Bytes through serial communication. In order to use it, we have to use the

Byte Pack to transform each data into 4 bytes and send all data in a row. The reason why we use byte pack is that it can make the receiver of the serial data easier to unpack and analyze the data.

2.5.2 Receiving serial Communication

The data sent by the DCM was broken down into bytes before being received by the serial communication subsystem.

Parameter Name	**Data Type**
Mode	uint16(2 bytes) -> 0: AOO; 1: VOO; 2: AAI; 3: VVI; 4: AOOR; 5: VOOR; 6: AAIR; 7: VVIR
LRL (Lower rate limit)	uint16(2 bytes)
URL(Upper rate limit)	uint16(2 bytes)
MSR	uint16(2 bytes)
Amplitude	uint16(2 bytes)
Pulse_Width	single(4 bytes)
Threshold	uint16(2 bytes)
ARP	single(4 bytes)
VRP	single(4 bytes)
AV_delay	single(4 bytes)
Activity_Threshold	single(4 bytes)
Reaction_time	uint16(2 bytes)
Response_factor	uint16(2 bytes)
Recovery_time	uint16(2 bytes)

This information was broken down in the serial receiving subsystem then carried to the main state flow module.

2.5.3 Sending Serial Communication

When Serial data is received in a certain pattern, the Simulink part will recognize the pattern send necessary data back into DCM part so that the DCM side can inspect the current status of the Pacemaker hardware.

More importantly, the current data of ATR_Signal and VENT_Signal will be sent back the DCM software part and generate a egram graph to actually see the pacing period and performance.

3. Current Progress and Modified Features.

3.1 Current Progress

3.1.1 Major Cyclic States of Pacing Process

The correct implementation of the 3 major cyclic states of pacing process can avoid critical errors that may in occur.

Charging C22: The capacitor (C22) is charged via the PWM signal input, which is controlled by the PACE_CHARGE_CTRL pin. The PWM input to the pin can vary the signal amplitude from 0-5V. (For example, input of 100 leads to signal amplitude of 5V)

Discharging Blocking Capacitor (C21): After AV pacing, the blocking capacitor should be discharged to prevent charge build-up, ensuring no harm is done to the patient. Discharging is done by grounding the ring of the recently paced chamber (ATR GND CTRL or VENT GND CTRL set to TRUE(1)). This state can be combined with the Charging C22 state. (Combined in current Simulink code)

AV PACING: Once the Pacing Capacitor (C22) is charged, the system may require that a pace be applied to either in the atrium or the ventricle. PACE_GND_CTRL remains high while turning off the corresponding GND_CTRL (ATR or VENT_GND_CTRL). As well as turning on the corresponding PACE_CTRL(ATR or VENT_PACE_CTRL).

3.1.2 AOO and VOO Modes:

Both modes have been successfully implemented with two states—Pacing and C22 Charge & C21 Discharge. These modes deliver continuous pacing signals to the respective chambers (atrium or ventricle), ensuring that heart chambers contract steadily without interruption from natural heart signals.

3.1.3 HeartView Testing:

Tests using the **HeartView application** confirmed that all modes are functioning as intended. The pacemaker complied as intended in all modes with reference to the natural heartbeat and rhythm.

3.1.4 AAI and VVI Modes:

Both modes are complete. Five states were defined for each mode:

Entry State: Initializes mode logic and prepares the system for pacing or sensing.

Charging State: Charges capacitor C22 for pacing operations.

Sensing State: Monitors natural heart activity through the FRONTEND_CTRL signal.

Sensed State: If a natural beat is detected, the pacemaker inhibits pacing.

Pacing State: If no natural activity is detected, the system delivers a pulse.

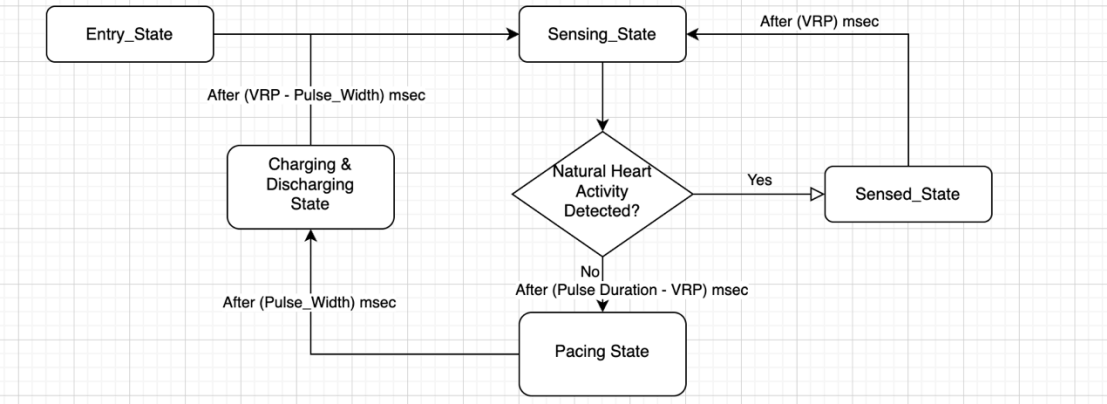


Figure 1: Simplified VVI stateflow

3.1.5 VRP Implementation: (ARP is Identical)

VRP (Ventricular Refractory Period) starts immediately after a ventricular event (paced or natural). Current implementation ensures proper timing delays between VRP and Pulse Width. For VVI mode, VRP = 500 ms ensures the system does not respond to spurious signals during recovery phases. However, an issue was identified during testing when pacing persisted despite the natural heart rate exceeding 150 BPM.

3.2 Modified Features from assignment 1.

3.2.1 Improved safety measures in both frontend and backend

Better safety measures are implemented in both frontend and backend than the previous versions. On the backend side, which is the Simulink code side, a limit is implemented in parameters input using comparator blocks to ensure the input will not exceed the certain range shown in the requirements.

3.2.2 PWM Sensitivity threshold modified.

In previous implementation, a sensitivity threshold of 70 has been used. Now it's modified to 66 to have a better and more accurate performance. The threshold 66 has been tested through many test cases to ensure a better performance.

3.2.3 Debug and Error handling Method update

In the present version of Simulink code, we used the different LEDs on the given hardware board to implement better debugging methods. Through the lighting up of different LEDs, we can tell which chamber is being paced and which mode the pacemaker is in without using any other methods, which makes the debugging process easier.

3.3 Changes in Requirements and Design Decisions

3.3.1 Changes in Requirements

With the moving on in the Pacemaker project, more requirements are needed to be implemented and considered, changes should be applied as well.

One change is that UART serial communication method should be introduced in order to make the pacemaker controllable through the DCM interface. The DCM interface is introduced to manage the modes and all programmable parameters in the pacemaker backend. With the serial communication method introduced, the pacemaker will become more user-friendly and accessible.

The other change is that the safety measures in the pacemaker should be further improved and refined because now that more modes has been introduced, more scenarios and situations are covered, there are even modes that require pacing in both chambers, which indicates there will

be more danger is the current safety measures goes wrong. Therefore, the safety measures should be further refined.

3.3.2 Changes in Design Decisions

With more pacing modes introduced to the pacemaker, the main state flow should be expanded further to integrate the new modes. The whole project also needs new modules to implement the new modes and features, such as: Serial communication module and Rate adaptive module. To implement all these new modules, new input parameters will be introduced, new logic blocks and state flows are needed.

In order to incorporate the new serial communication modules for both COM_IN and COM_OUT, we need to use the UART ports, we also have to implement new logics to analyze the incoming serial communication. we also have to use more wirings to make sure the setting data coming in will be correctly brought to various input ports.

In brief, the development of the pacemaker requires more variables and logic blocks, which may further affect the main state flow to incorporate and adjust new features.

3.4 Bonus Mode: DDDR Mode

The DDDR pacing mode is a dual-chamber, rate-adaptive pacemaker mode designed to provide optimal cardiac pacing by maintaining atrioventricular (AV) synchrony and adjusting the pacing rate based on physiological needs.

Key Features:

Dual-Chamber Pacing: Paces both the atrium and ventricle as needed.

Dual-Chamber Sensing: Senses intrinsic electrical activity in both the atrium and ventricle.

Dual Response: Inhibits pacing if intrinsic activity is detected or ButtonPress Detected. Triggers ventricular pacing in response to sensed atrial activity after a programmed AV delay.

Rate Adaptation:Adjusts pacing rate dynamically based on patient activity using sensors.

Programmable Parameters

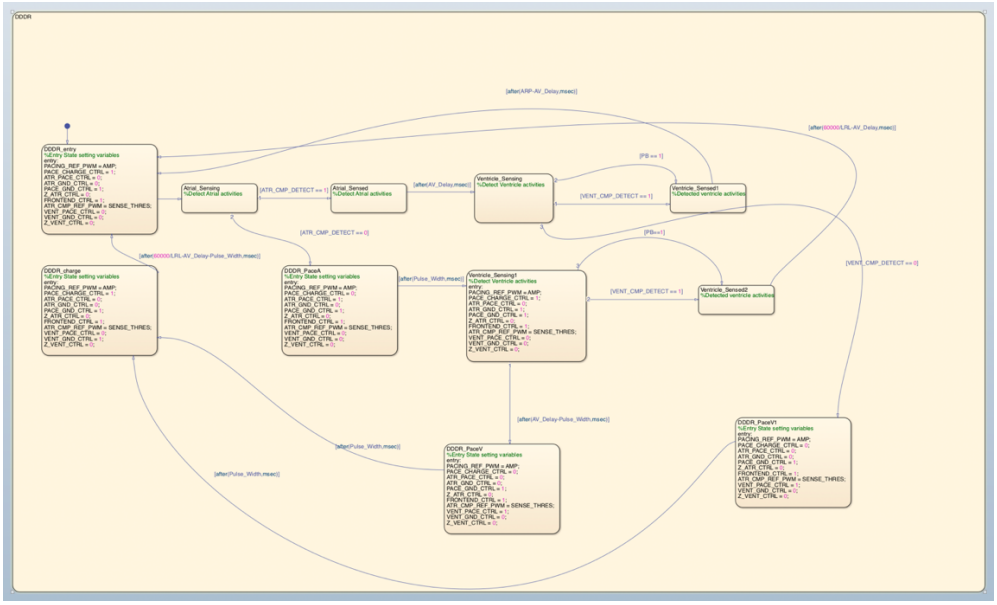
Lower Rate Limit (LRL): Minimum pacing rate (e.g., 60 bpm).

Maximum Sensor Rate (MSR): Maximum pacing rate during activity (e.g., 120 bpm).

AV Delay: Time between atrial and ventricular events (e.g., 150 ms).

Response Factor (RF): Sensitivity to changes in activity level (e.g., 1–16).

Reaction and Recovery Times: Control how quickly the pacing rate increases or decreases.



DDDR Implementation

4. Safety Assurance

This section identifies aspects of the design implemented in Simulink to ensure user safety.

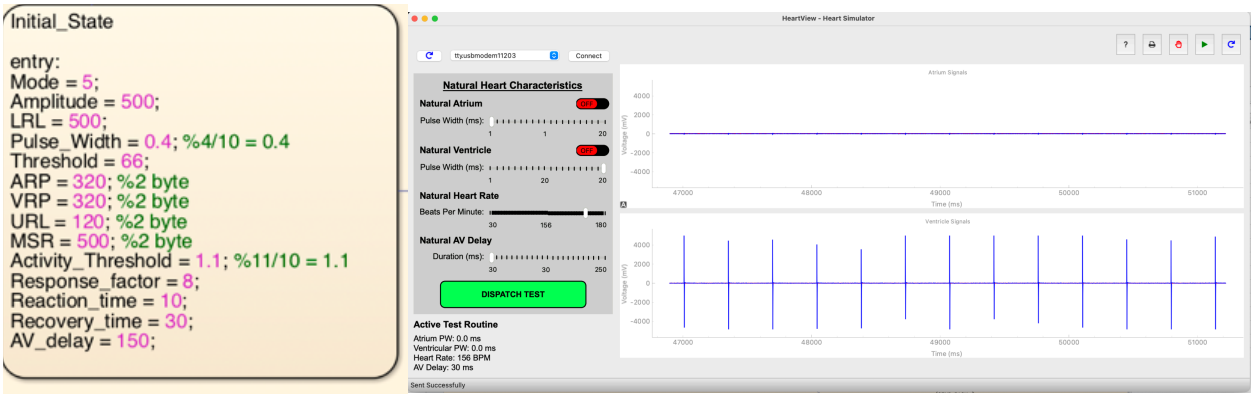
4.1. Approach

Within Simulink the main approach to safety was to prevent any extremely large pacing values from being sent to the pacemaker. As a backup layer to the additional safety standards completed by the DCM team.

4.2. Standards

Within Simulink the main approach to safety was to prevent any extremely large pacing values from being sent to the pacemaker. Mainly as a backup layer to the additional safety standards completed by the DCM team.

4.3. Testing



Inputs to VOOR and the resulting output, for a safety assurance test.

As can be seen, despite an LRL value of 500, with an MSR value of 500 as well, the pacemaker remained stable at a pace of 175, which is the software limit set on the pacemaker.

5. Testing and results

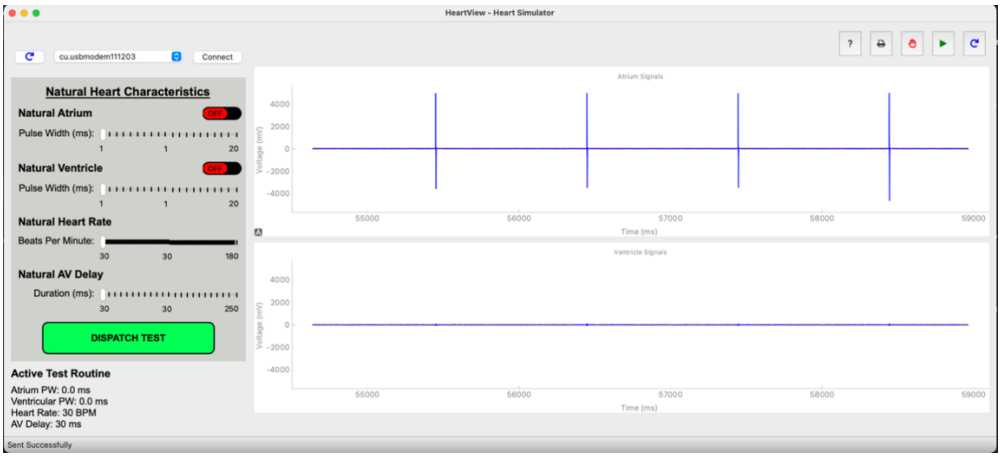
5.1. AOO Testing:

Test Case: NHR (Natural Hear Rate): 30 BPM, 60 BPM pace setting, random natural atrial activity.

Purpose: Basic functionality test, ensure that the pacemaker works as intended in AOO.

Expected Performance: Constant 60 BPM pacing pulse regardless of natural atrial activity.

Result: Pass

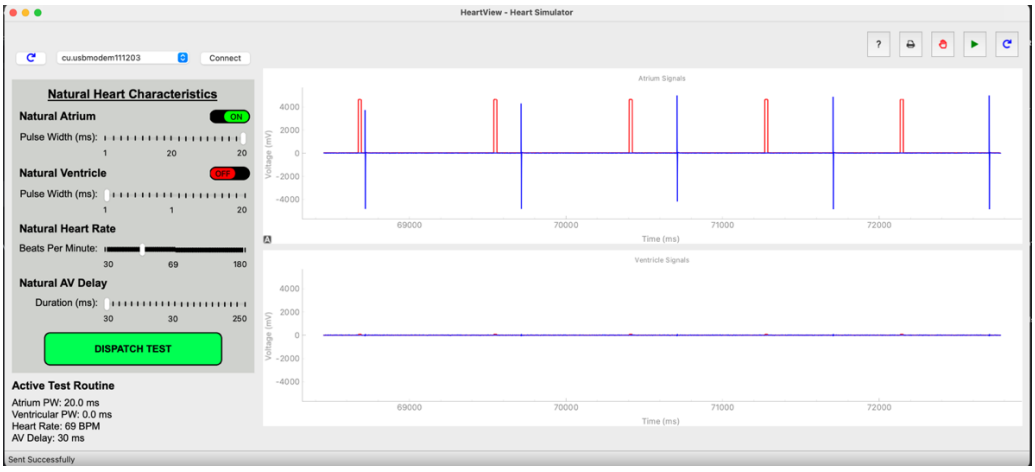


Test Case: NHR: 70 BPM, 60 BPM pace setting, random natural atrial activity.

Purpose: Ensure that the pacemaker paces no matter what the NHR is.

Expected Performance: Constant 60 BPM pacing pulse regardless of natural atrial activity.

Result: Pass

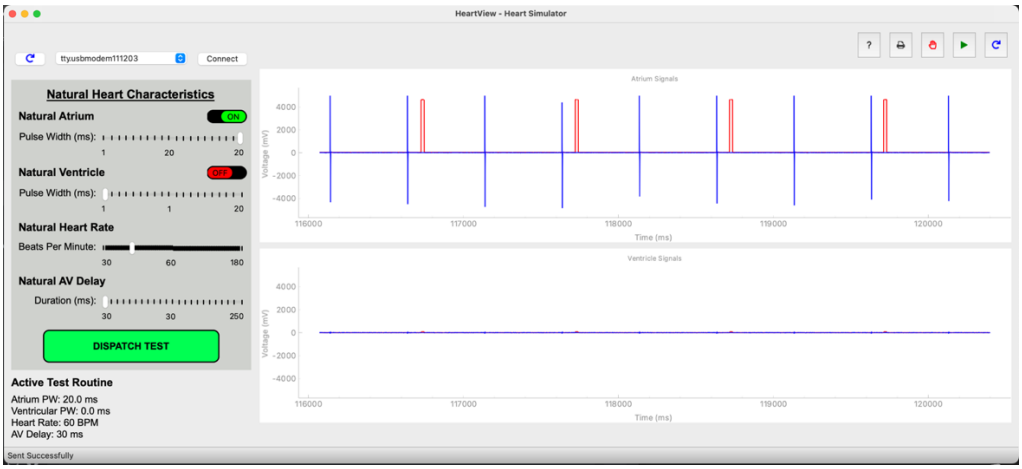


Test Case: NHR: 60 BPM, 120BPM pace setting, random natural atrial activity.

Purpose: Ensure that a higher pace setting still works as intended.

Expected Performance: Constant 120 BPM pacing pulse regardless of natural atrial activity.

Result: Pass



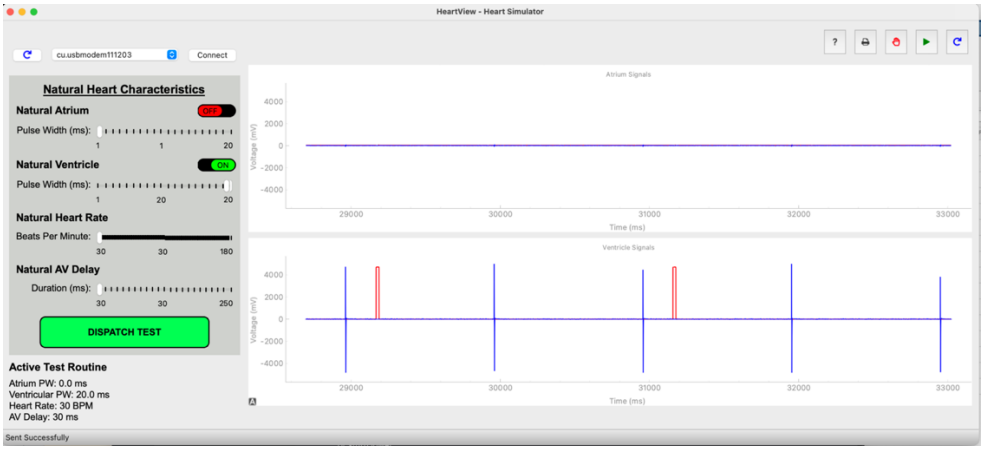
5.2. VOO Testing:

Test Case: NHR (Natural Hear Rate): 30 BPM, 60 BPM pace setting, random natural ventricle activity.

Purpose: Basic functionality test, ensure that the pacemaker works as intended in VOO.

Expected Performance: Constant 60 BPM pacing pulse regardless of natural ventricle activity.

Result: Pass

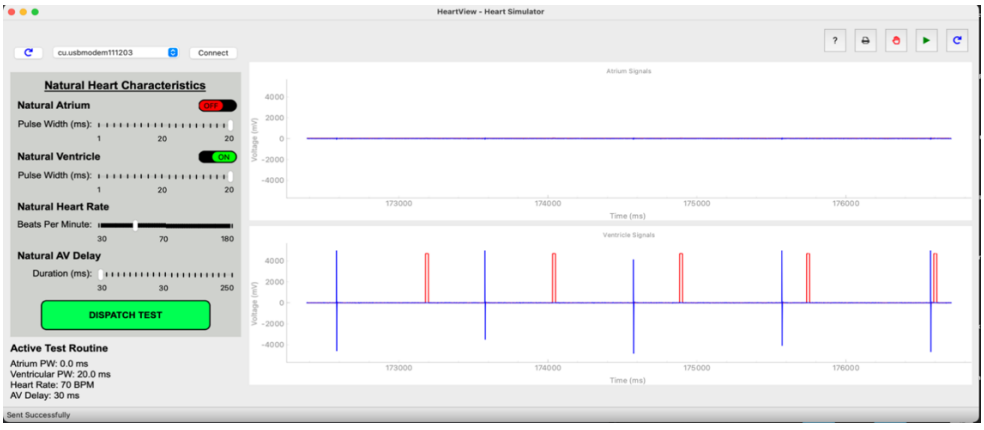


Test Case: NHR: 70 BPM, 60 BPM pace setting, random natural ventricle activity.

Purpose: Ensure that the pacemaker paces no matter what the NHR is.

Expected Performance: Constant 60 BPM pacing pulse regardless of natural ventricle activity.

Result: Pass

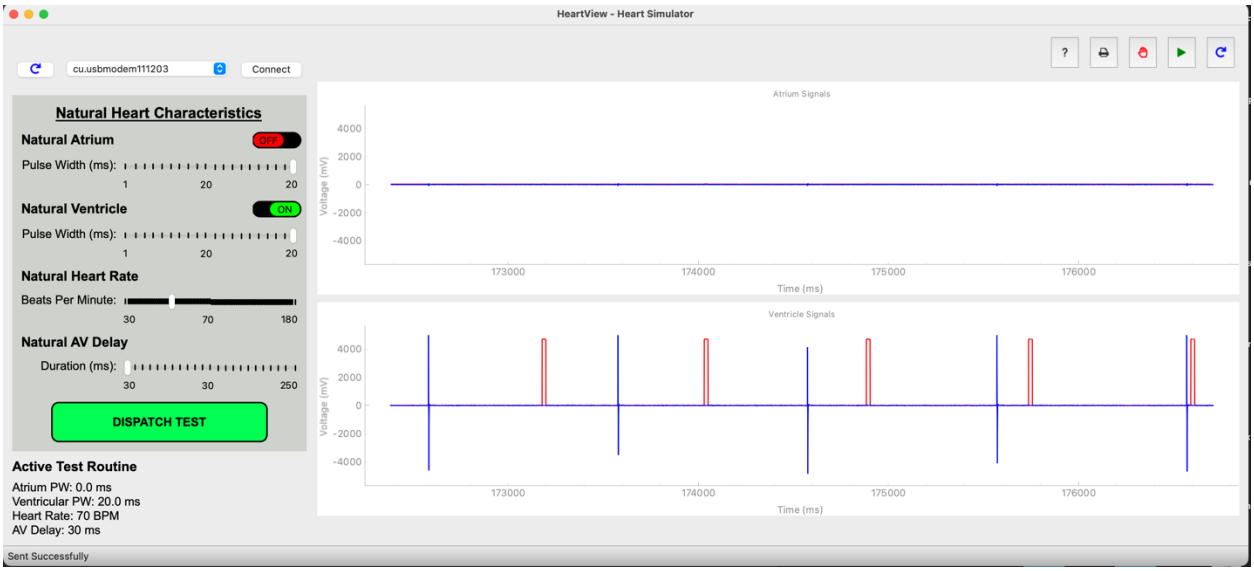


Test Case: NHR: 70 BPM, 120 BPM pace setting, random natural ventricle activity.

Expected Performance: Constant 120 BPM pacing pulse regardless of natural ventricle activity.

Purpose: Ensure that a higher pace setting still works as intended.

Result: Pass



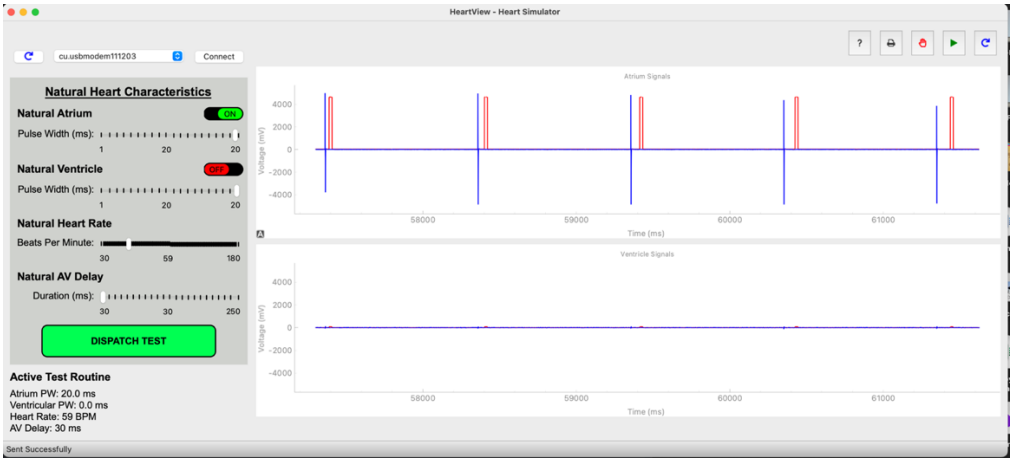
5.3. AAI Testing:

Test Case: Natural Atrial activity 59 BPM, 60 BPM pace setting, ARP 320 msec.

Purpose: Ensure the basic functionality of AAI mode.

Expected Performance: Pacing pulse occurs before every natural atrial activity.

Result: Pass

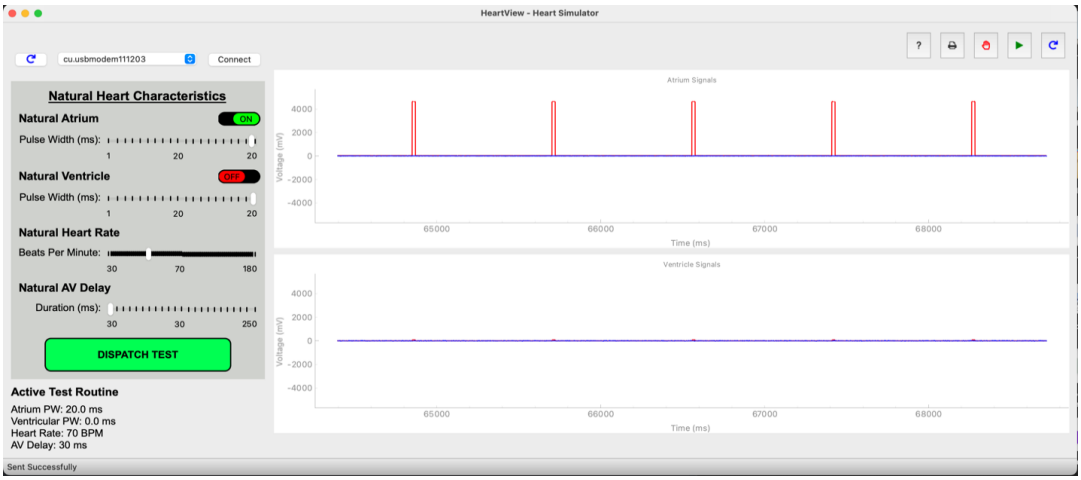


Test Case: Natural Atrial activity 70 BPM, 60 BPM pace setting, ARP 320 msec.

Purpose: Ensure that the AAI mode does not pace unless below the pace setting.

Expected Performance: Natural atrial activity of 70 BPM without any pacemaker activity.

Result: Pass

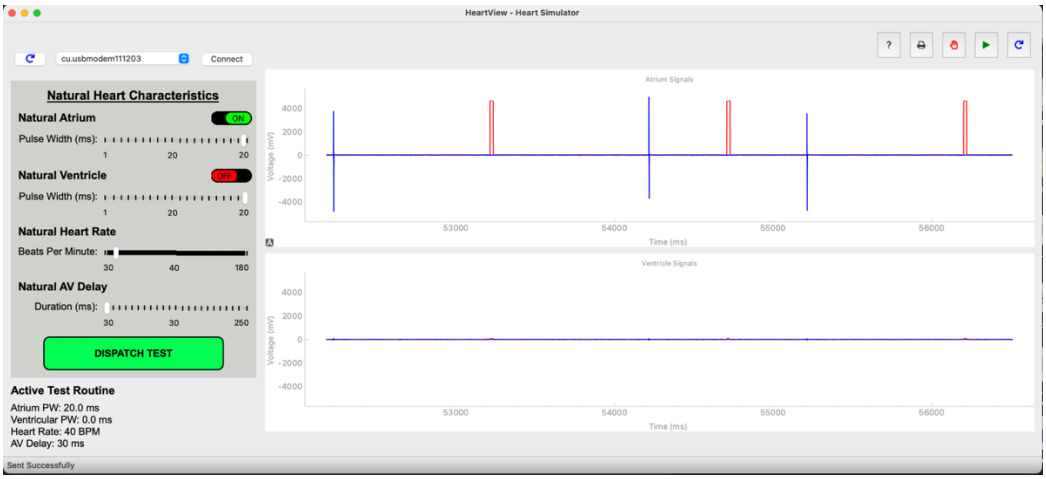


Test Case: Natural Atrial activity 40 BPM, 60 BPM pace setting, ARP 320 msec.

Purpose: Make sure that the pacemaker paces more often when natural atrial activity is not at set rate.

Expected Performance: Pacing pulse occurs between every natural atrial activity.

Result: Pass



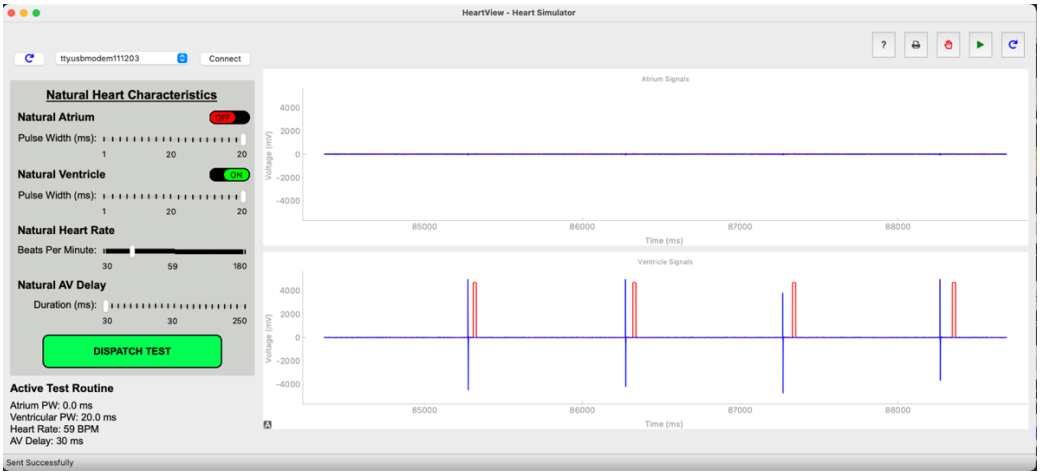
5.4. VVI Testing:

Test Case: Natural Ventricle activity 59 BPM, 60 BPM pace setting, VRP 320 msec.

Purpose: Basic functionality test, make sure VVI mode works.

Expected Performance: Pacing pulse occurs before every natural Ventricle activity.

Result: Pass

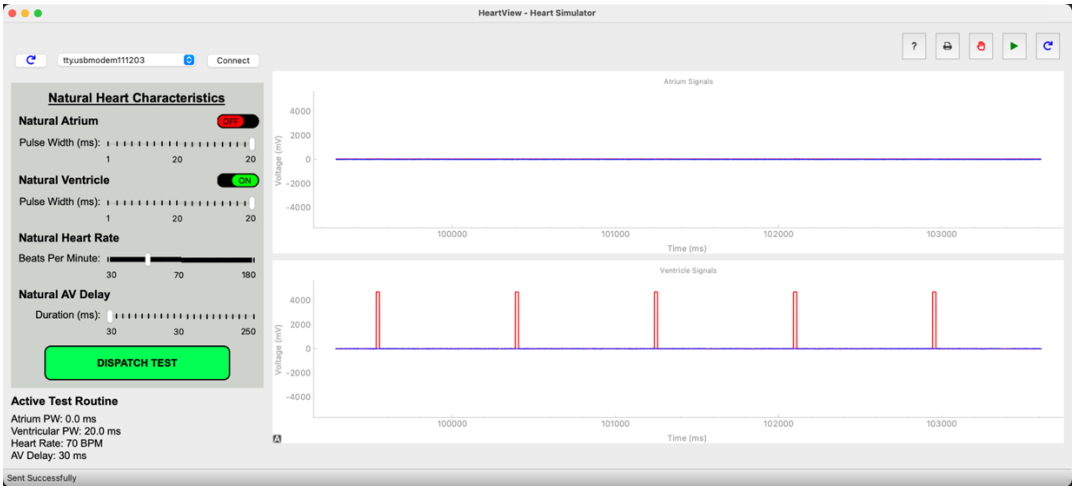


Test Case: Natural Ventricle activity 70 BPM, 60 BPM pace setting, VRP 320 msec.

Purpose: Ensure that the AAI mode does not pace unless below the pace setting.

Expected Performance: Natural Ventricle activity of 70 BPM without any pacemaker activity.

Result: Pass

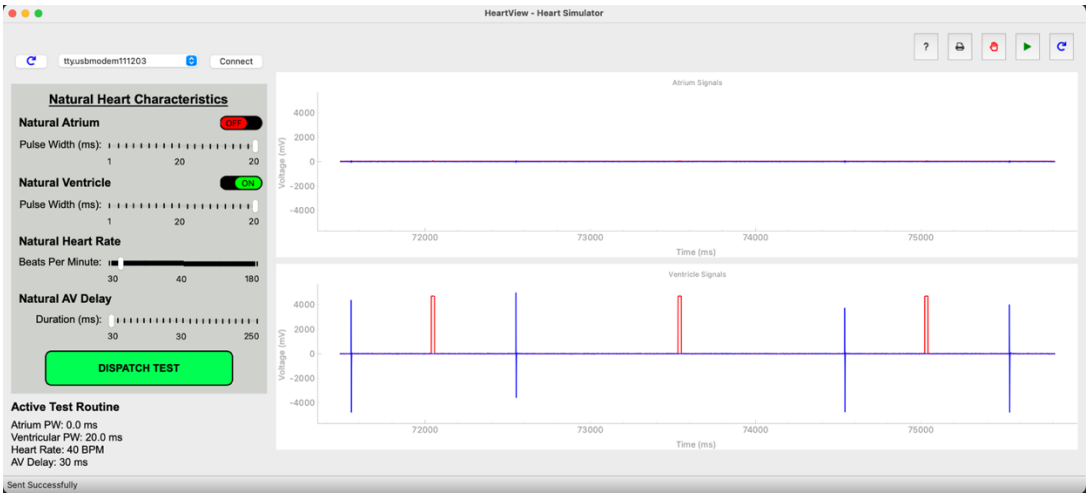


Test Case: Natural Ventricle activity 40 BPM, 60 BPM pace setting, VRP 320 msec.

Purpose: Make sure that the pacemaker paces more often when natural ventricle activity is not at set rate.

Expected Performance: Pacing pulse occurs between every natural Ventricle activity.

Result: Pass



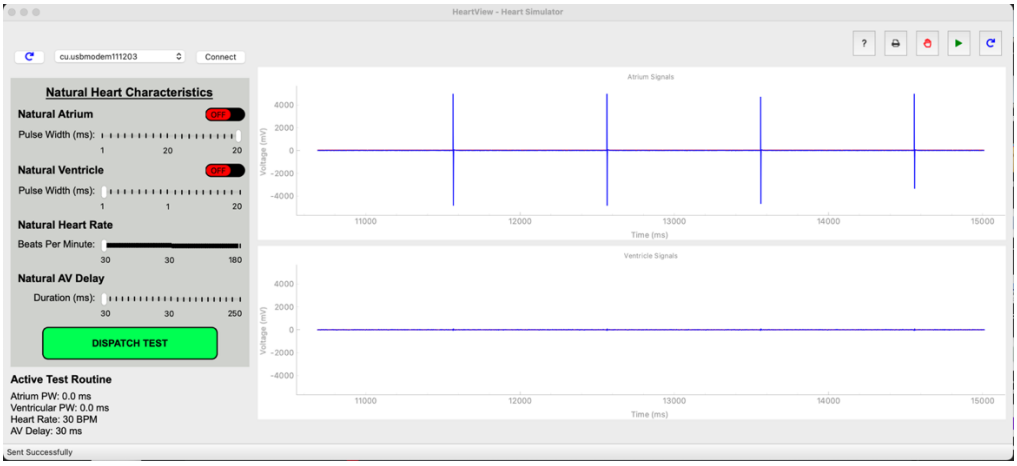
5.5. AOOR Testing:

Test Case: NHR (Natural Hear Rate): 30 BPM, 60 BPM pace setting, random natural atrial activity. No Motion.

Purpose: Ensure basic AOO functionality.

Expected Performance: Constant 60 BPM pacing pulse regardless of natural atrial activity.

Result: Pass

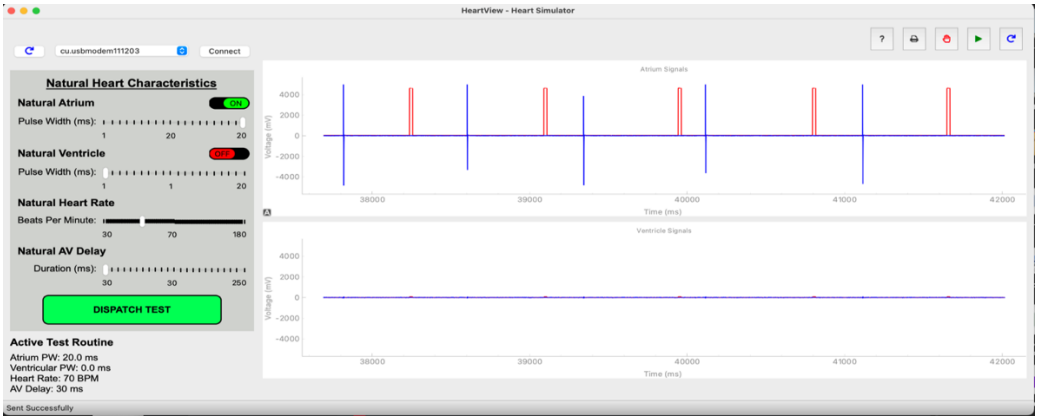


Test Case: NHR: 70 BPM, 60 BPM pace setting, random natural atrial activity. Some Motion.

Purpose: Ensure the basic rate adaptivity functionality works.

Expected Performance: Slow increase of pacing rate while in motion to above 60 BPM, up to 20 BPM higher.

Result: Pass

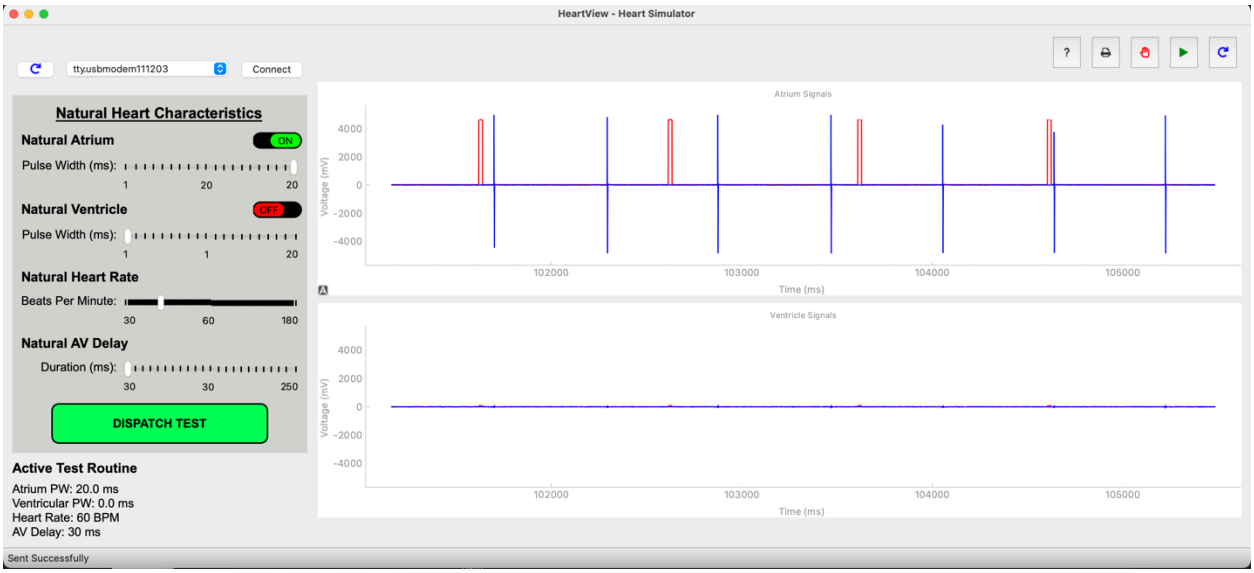


Test Case: NHR: 60 BPM, 80 BPM pace setting, random natural atrial activity. Lots of motion.

Purpose: Ensure that more motion results in a higher pacing rate.

Expected Performance: Slow increase of pacing rate while in motion to above 100 BPM up to 120BPM.

Result: Pass



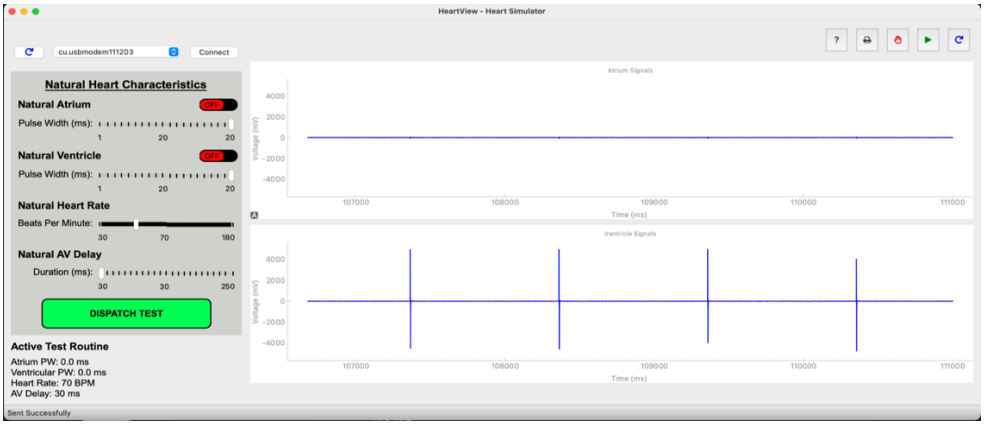
5.6. VOOR Testing:

Test Case: NHR (Natural Hear Rate): 70 BPM, 60 BPM pace setting, random natural ventricle activity. No Motion.

Purpose: Ensure basic VOO functionality.

Expected Performance: Constant 60 BPM pacing pulse regardless of natural ventricle activity.

Result: Pass

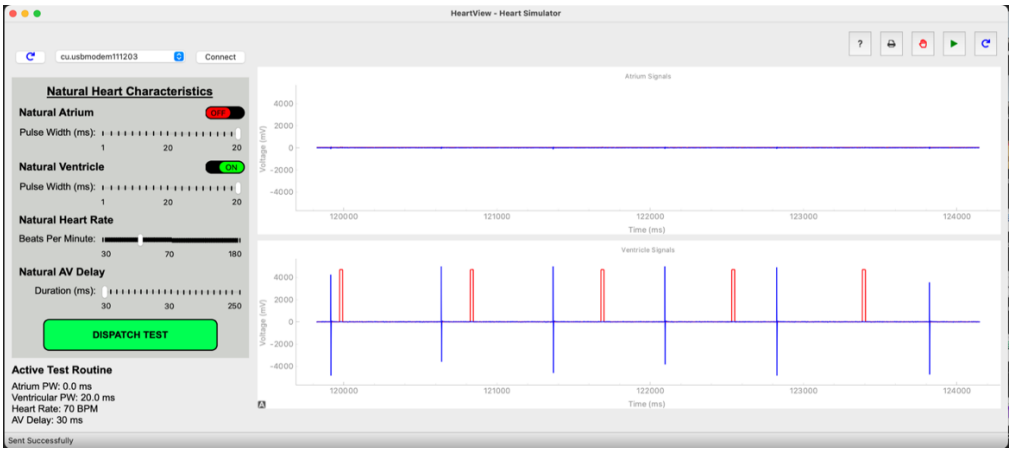


Test Case: NHR: 70 BPM, 60 BPM pace setting, random natural ventricle activity. Some Motion.

Purpose: Ensure the basic rate adaptivity functionality works.

Expected Performance: Slow increase of pacing rate while in motion to above 60 BPM, up to 20 BPM higher.

Result: Pass

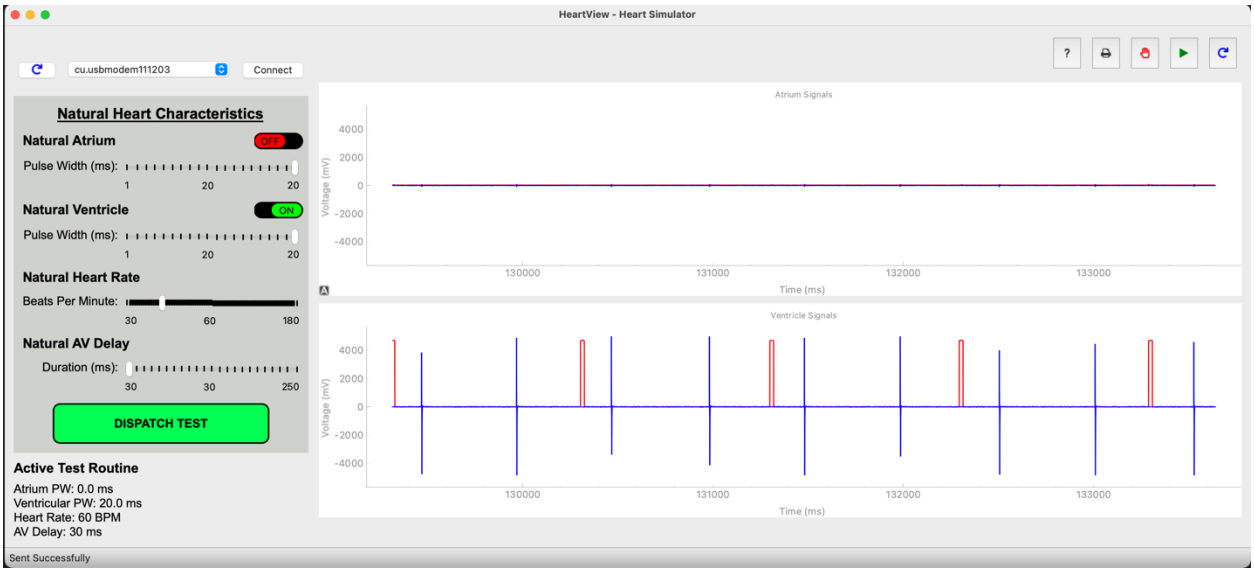


Test Case: NHR: 60 BPM, 80 BPM pace setting, random natural ventricle activity. Lots of motion.

Purpose: Ensure that more motion results in a higher pacing rate.

Expected Performance: Slow increase of pacing rate while in motion to above 100 BPM up to 120BPM higher.

Result: Pass



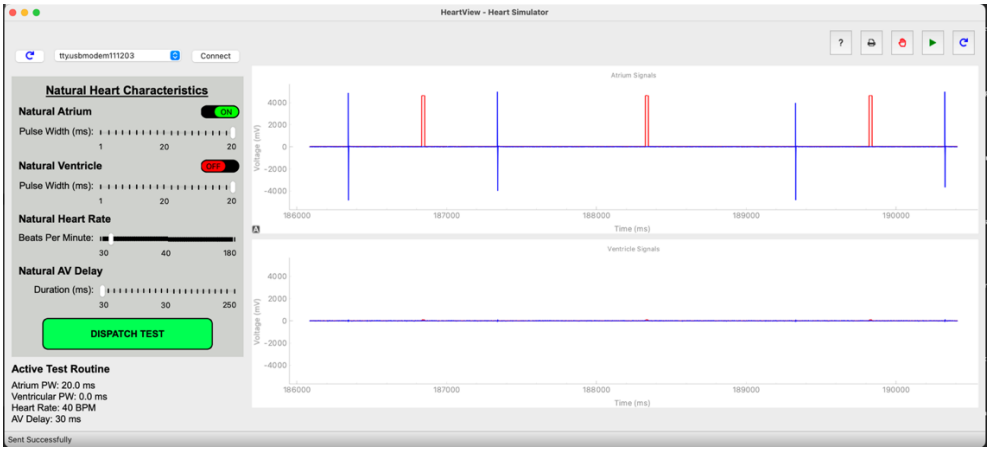
5.7. AAIR Testing:

Test Case: NHR: 40 BPM, Pacing Rate: 60, No motion.

Purpose: Ensure AAI performance even in AAIR mode.

Expected Performance: Pacing pulse occurs before every two natural atrium activities.

Result: Pass

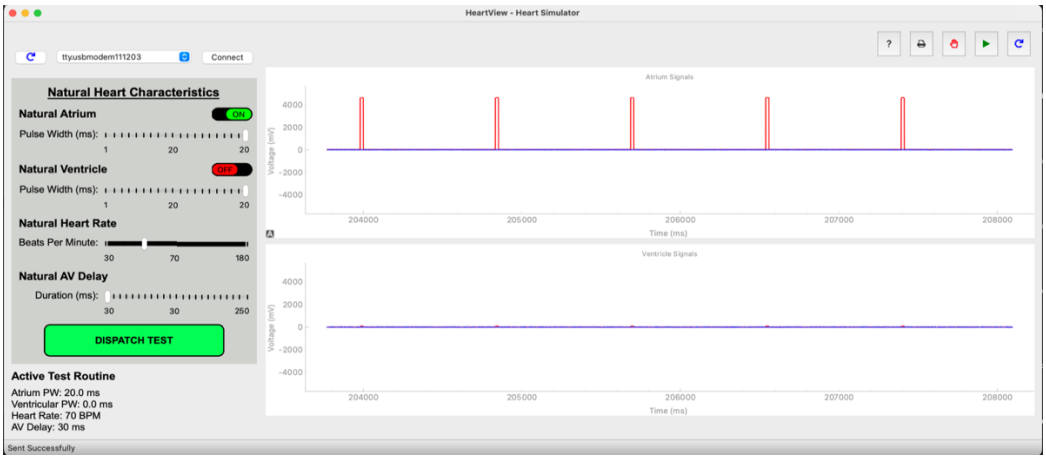


Test Case: NHR: 70 BPM, Pacing Rate: 60 BPM, No motion.

Purpose: Once again, ensure AAI performance even in AAIR mode.

Expected Performance: Natural atrial activity of 70 BPM without any pacemaker activity.

Result: Pass

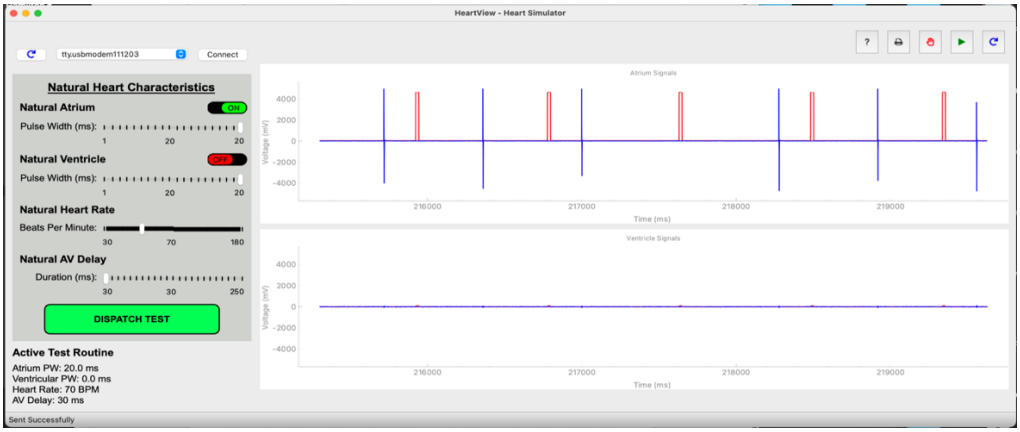


Test Case: NHR: 70 BPM, Pacing Rate: 60 BPM, with motion.

Purpose: Ensure rate adaptive mode increases pulse even if NHR is larger than the pacing rate, as long as there is motion.

Expected Performance: Pacing pulse occurs before every two natural atrium activities. As the increased motion increases the pacing rate, up to a limit of 120 BPM.

Result: Pass

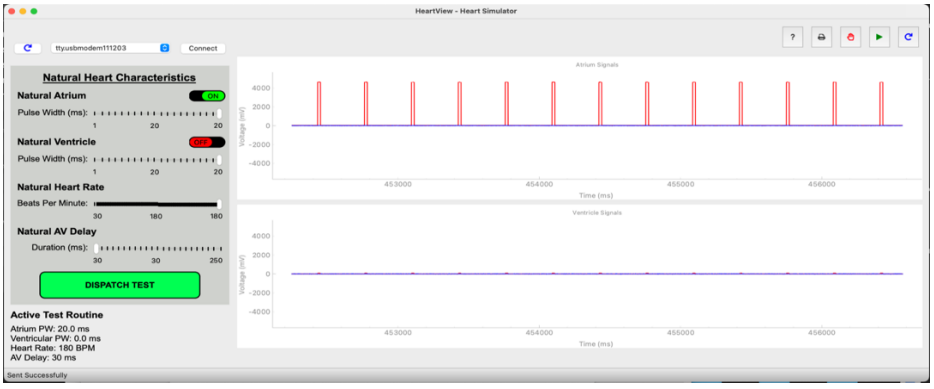


Test Case: NHR: 180 BPM, Pacing Rate: 60 BPM, with motion.

Purpose: Ensure that given high enough NHR, the pacing rate will not change over a certain limit, in this case MSR.

Expected Performance: No pacing no matter the ferocity of the motion, as the highest value of the adapted rate, or the MSR, is 120 BPM. Which is lower than the NHR.

Result: Pass



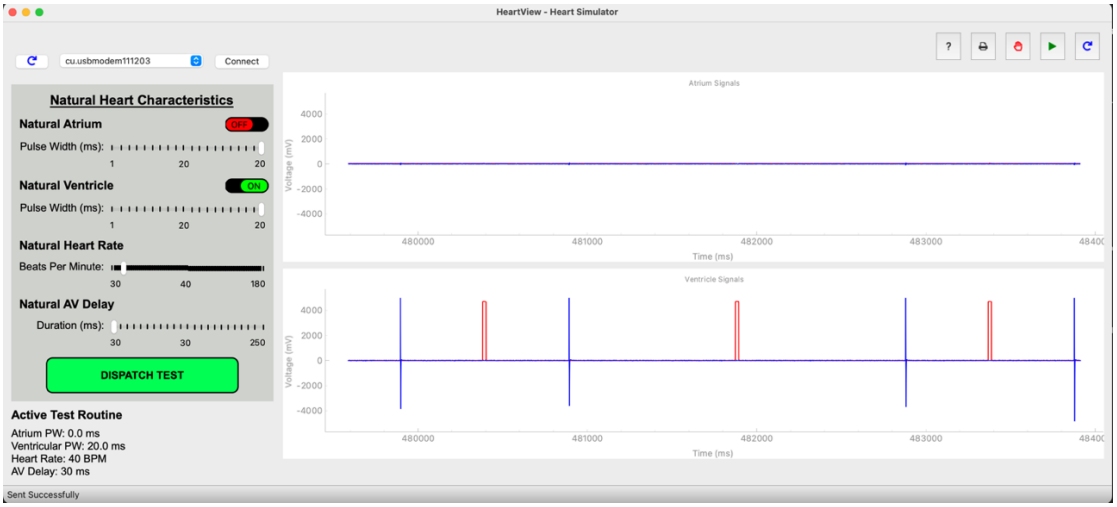
5.8. VVIR Testing:

Test Case: NHR: 40 BPM, Pacing Rate: 60, No motion.

Purpose: Ensure VVI performance even in VVIR mode.

Expected Performance: Pacing pulse occurs before every two natural ventricle activity.

Result: Pass

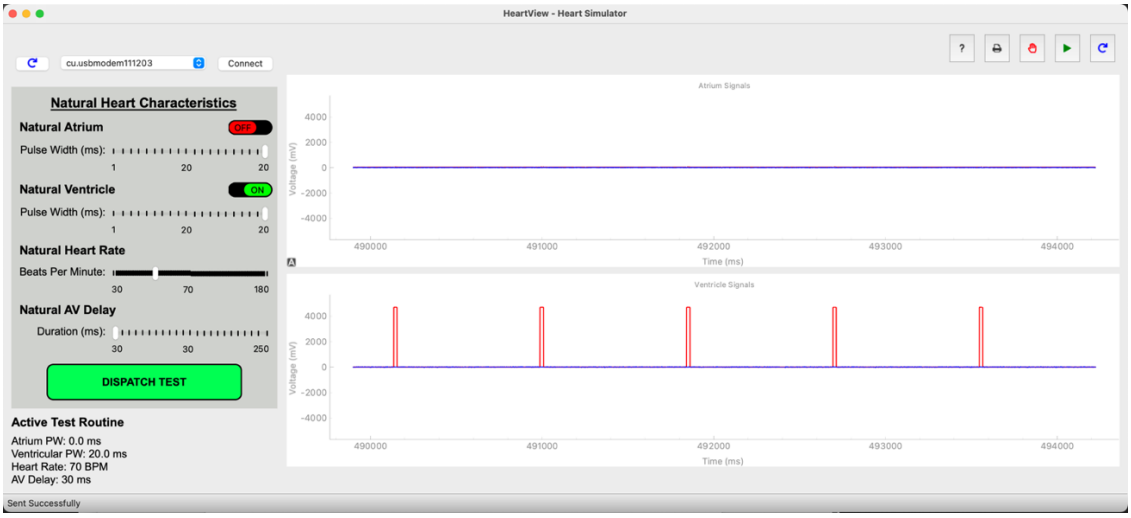


Test Case: NHR: 70 BPM, Pacing Rate: 60 BPM, No motion.

Purpose: Once again, ensure VVI performance even in VVIR mode.

Expected Performance: Natural ventricle activity of 70 BPM without any pacemaker activity.

Result: Pass

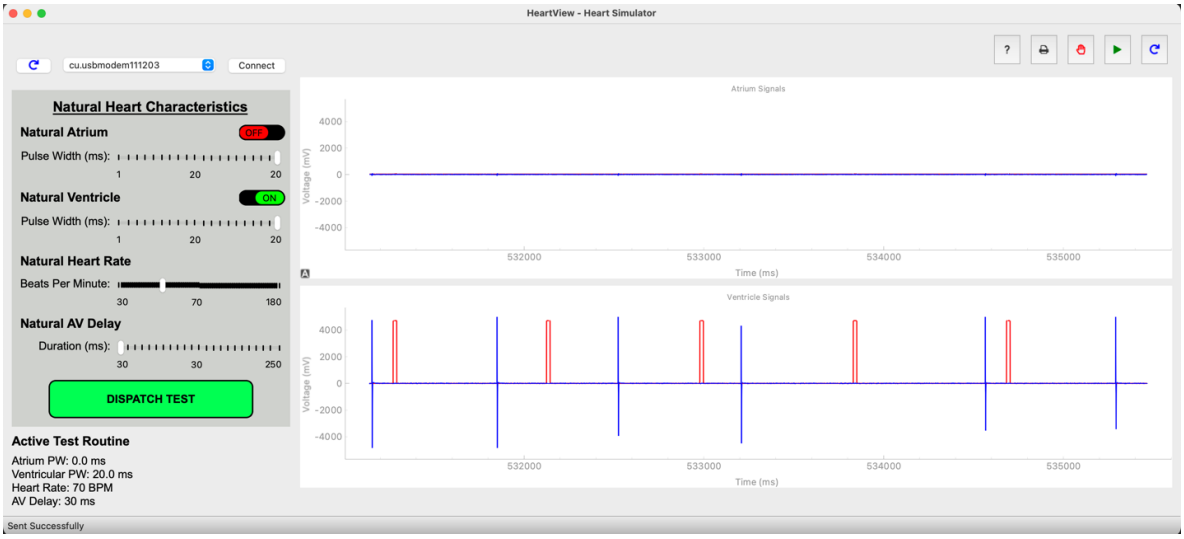


Test Case: NHR: 70 BPM, Pacing Rate: 60 BPM, with motion.

Purpose: Ensure rate adaptive mode increases pulse even if NHR is larger than the pacing rate if there is motion.

Expected Performance: Pacing pulse occurs before every two natural ventricle activity. As the increased motion increases the pacing rate, up to a limit of 120 BPM.

Result: Pass

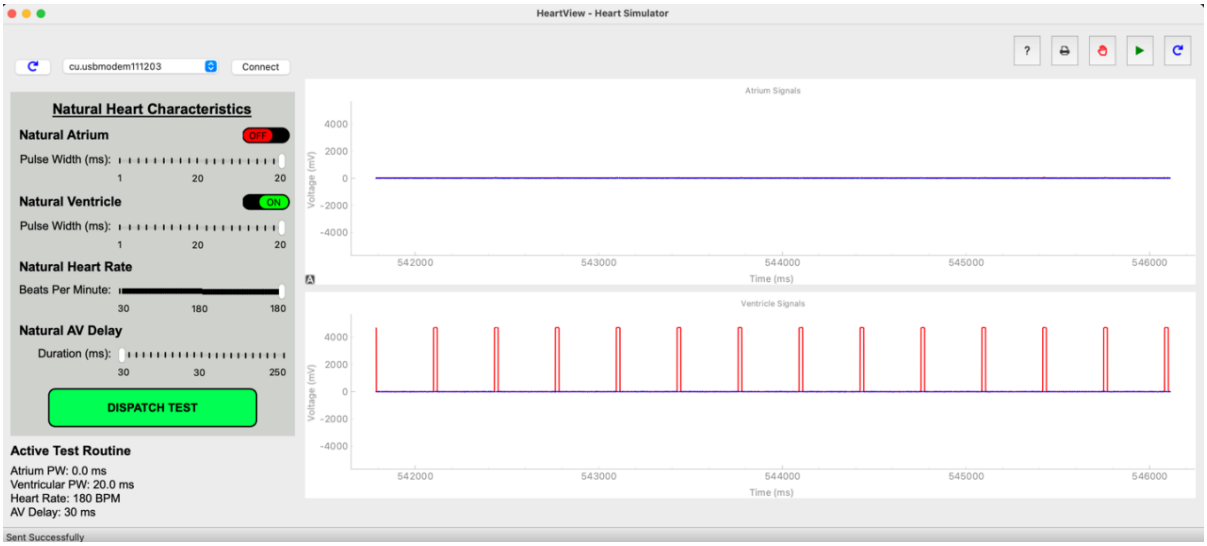


Test Case: NHR: 180 BPM, Pacing Rate: 60 BPM, with motion.

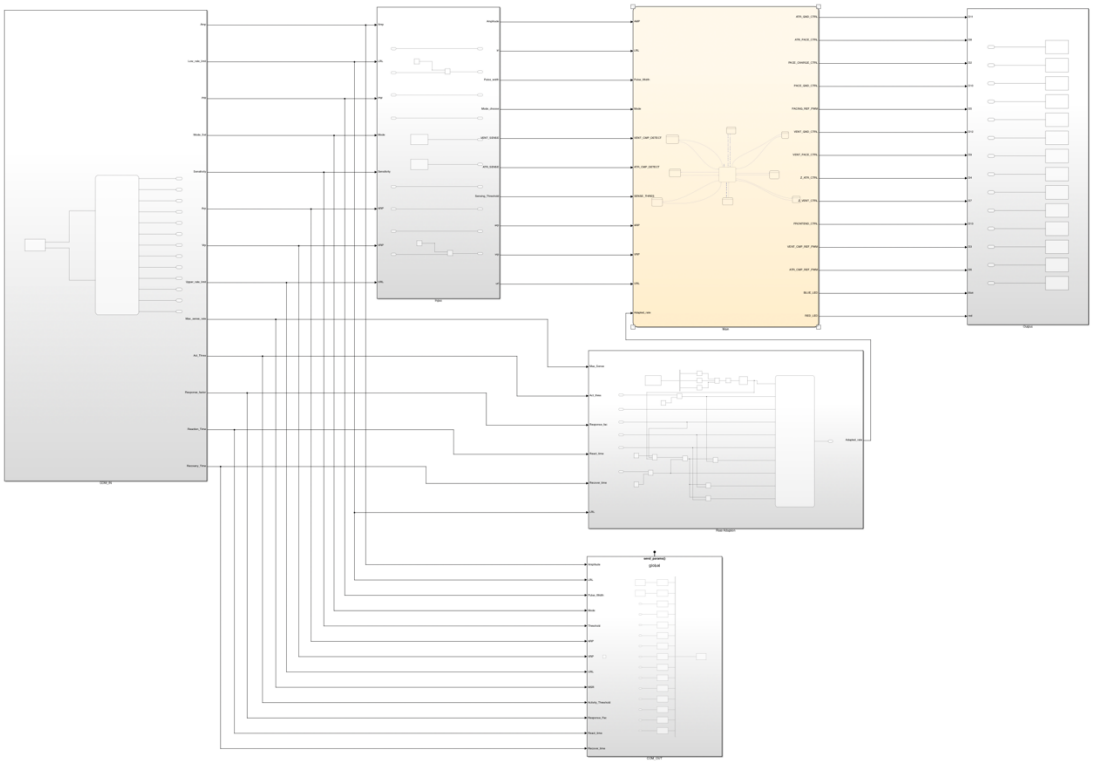
Purpose: Ensure that given high enough NHR, the pacing rate will not change over a certain limit, in this case MSR.

Expected Performance: No pacing no matter the ferocity of the motion, as the highest value of the adapted rate, or the MSR, is 120 BPM. Which is lower than the NHR.

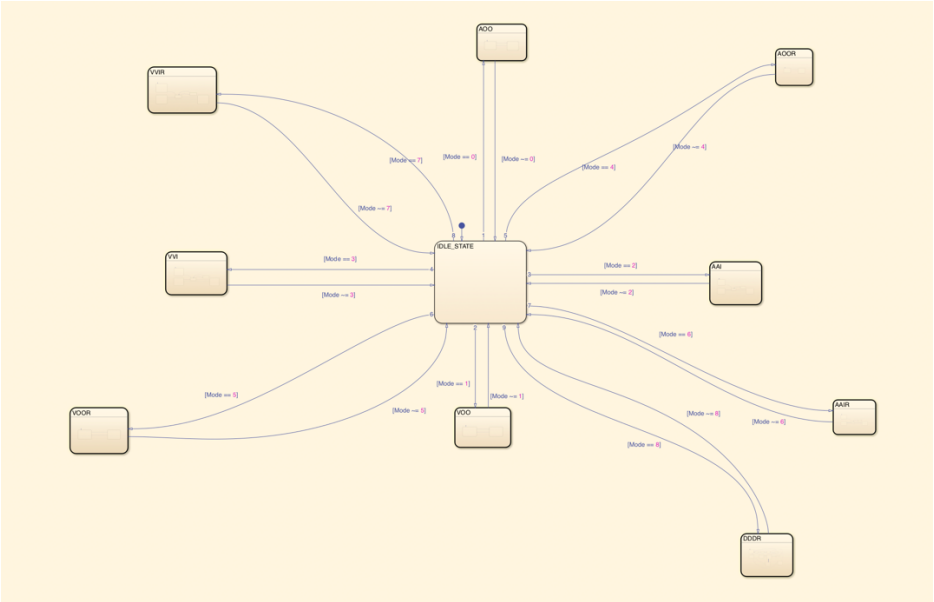
Result: Pass



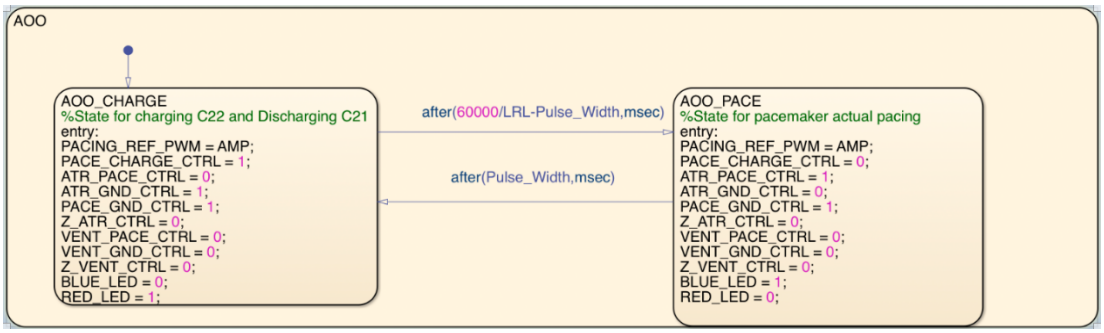
6. Simulink Model Screenshots



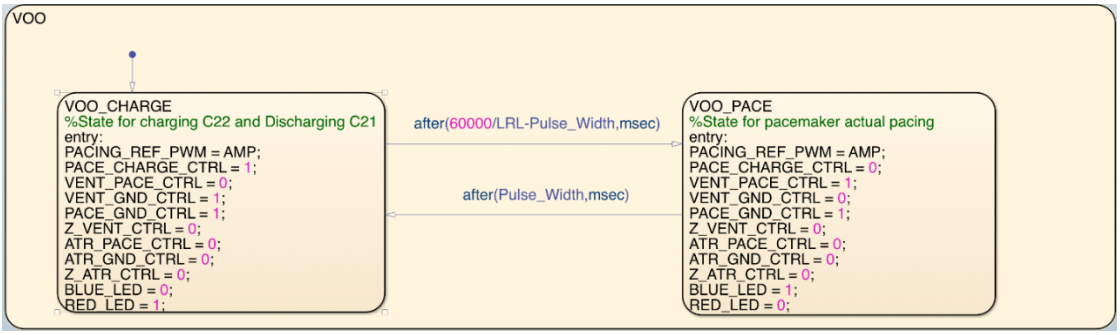
Screenshot of whole file



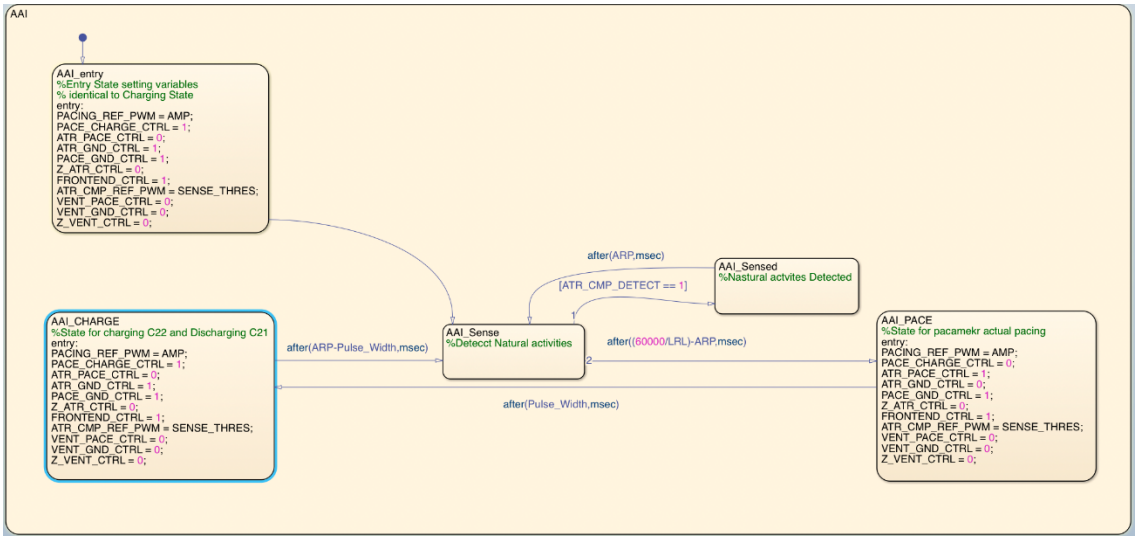
Main State flow



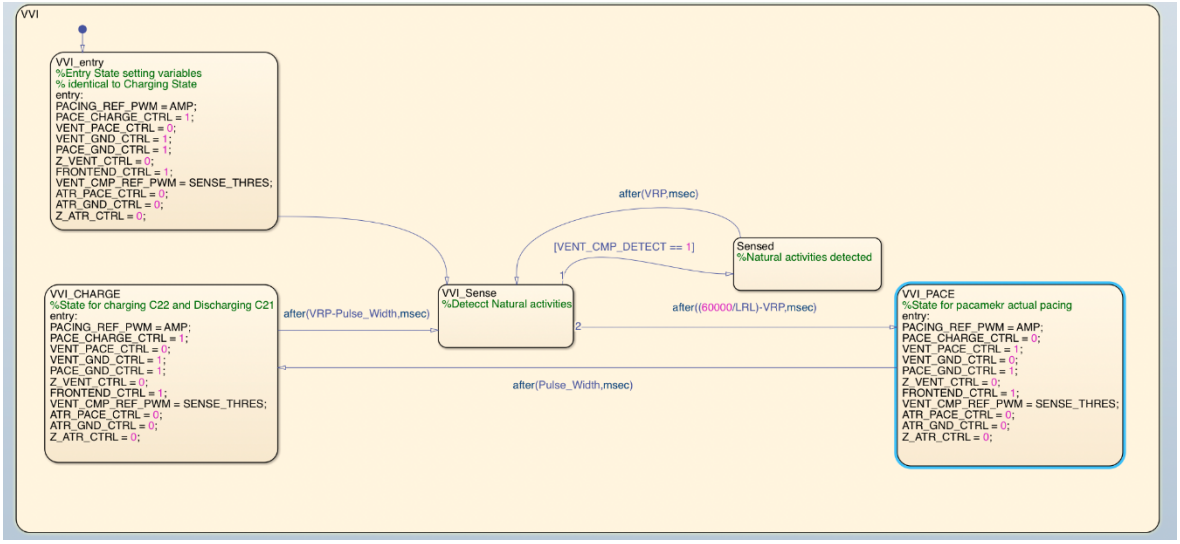
AOO mode



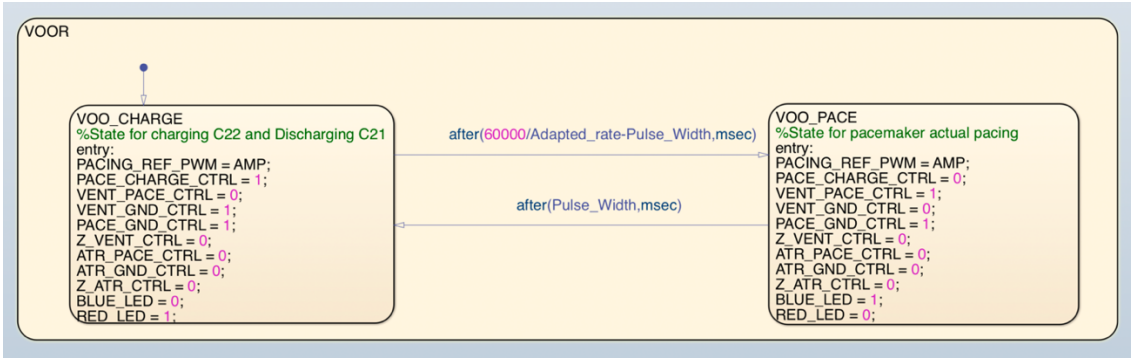
VOO mode



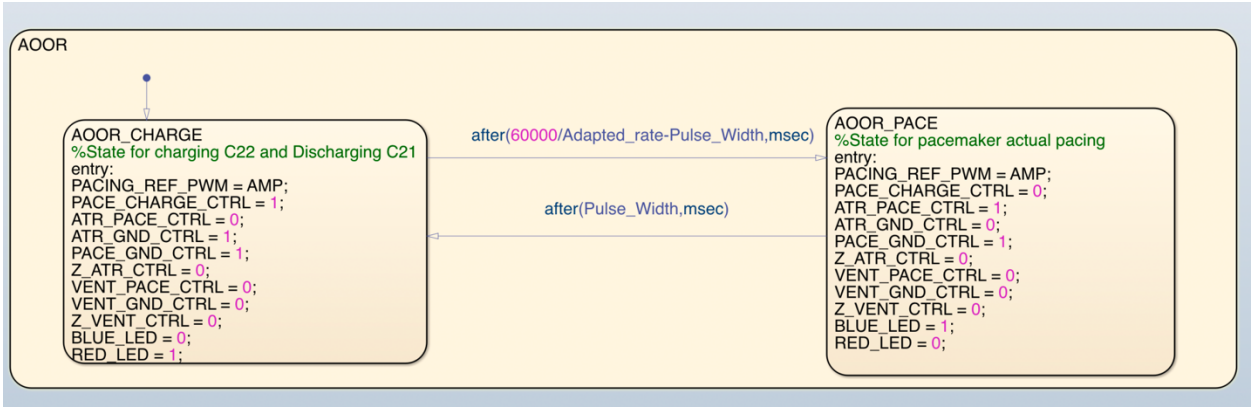
AAI mode



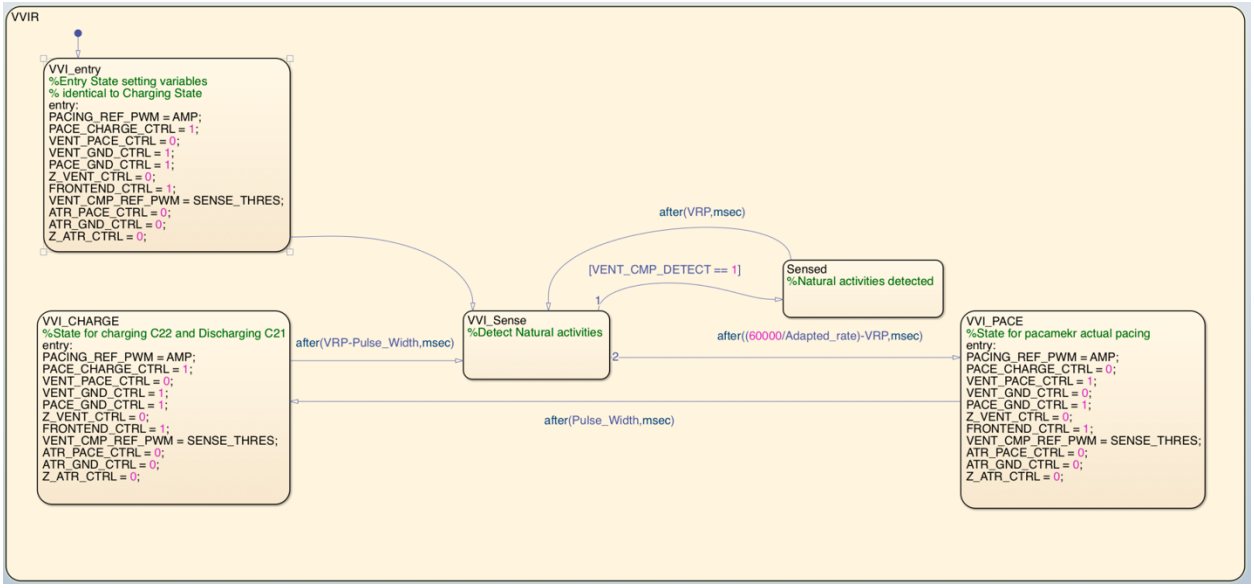
VVI mode



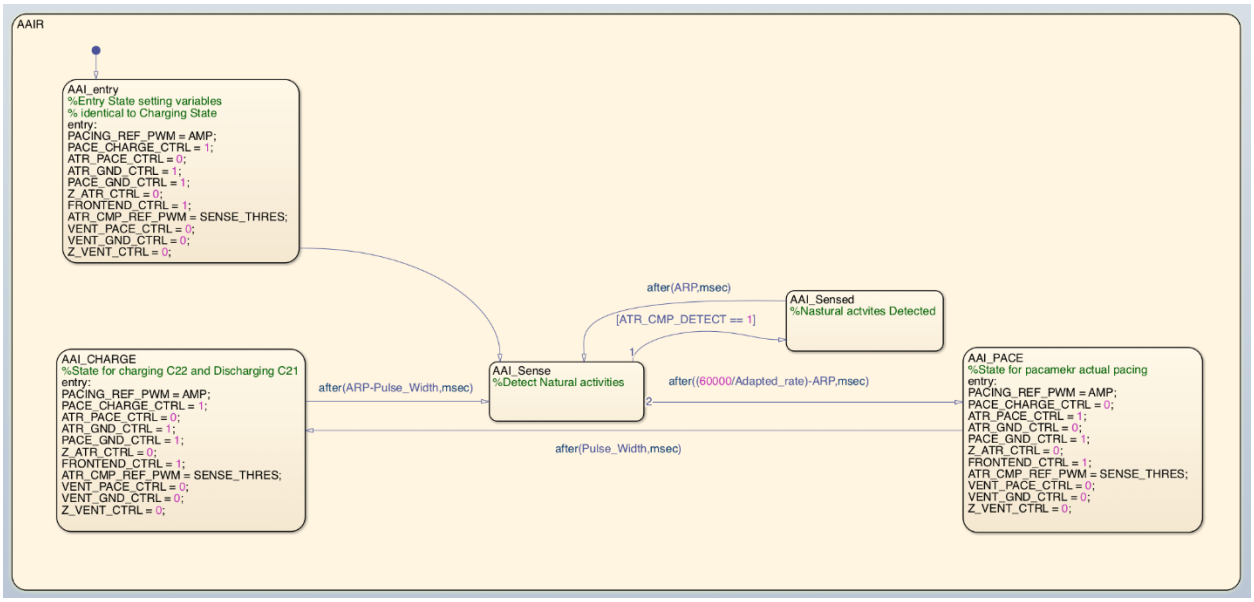
VOOR Mode



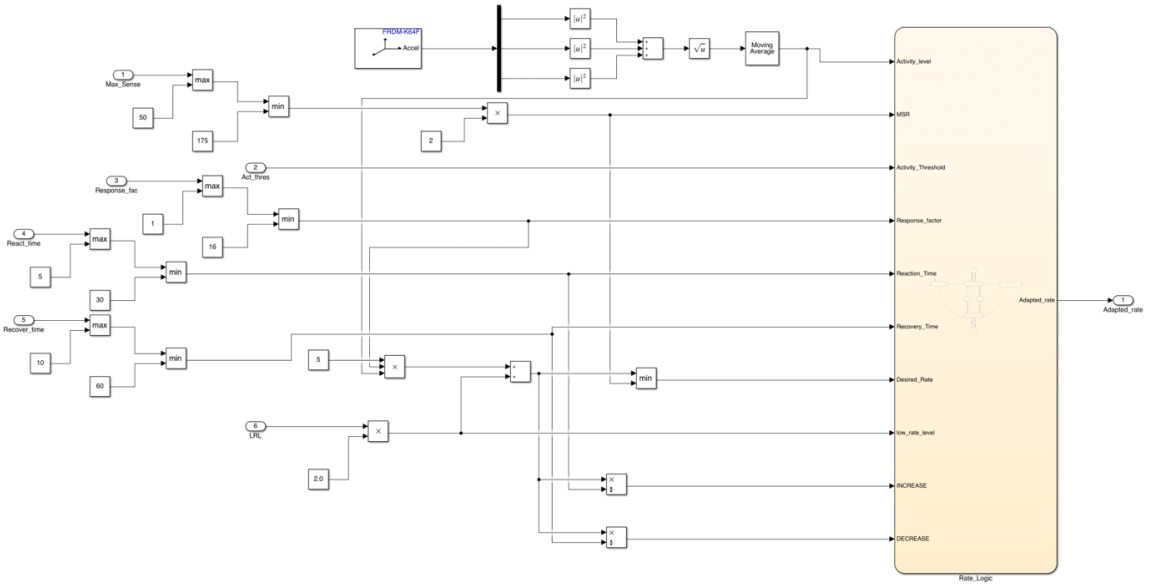
AOOR Mode



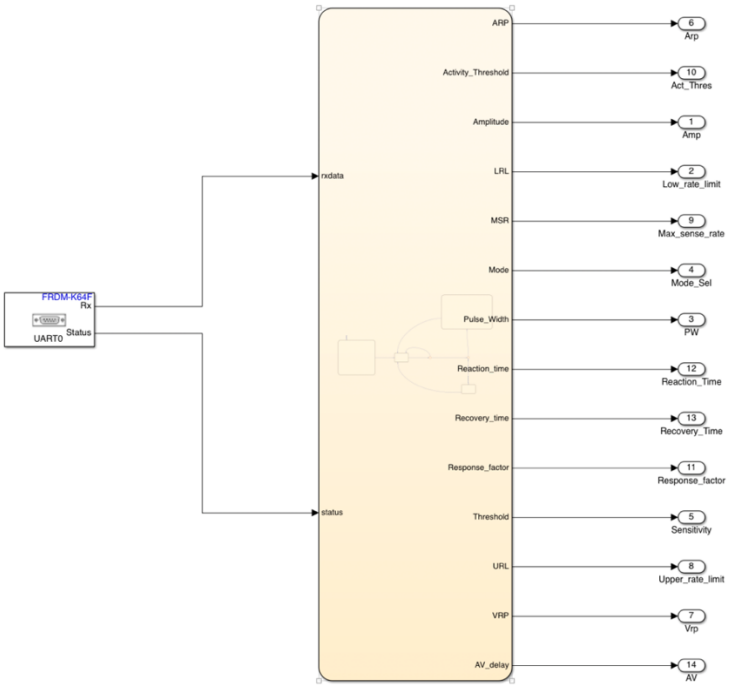
VVIR Mode



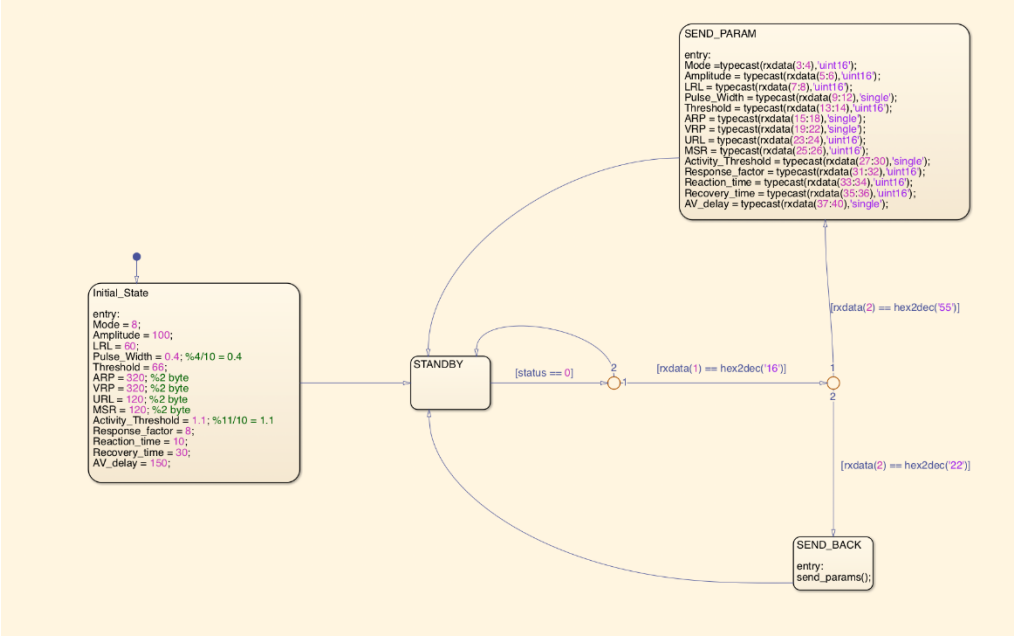
AAIR Mode



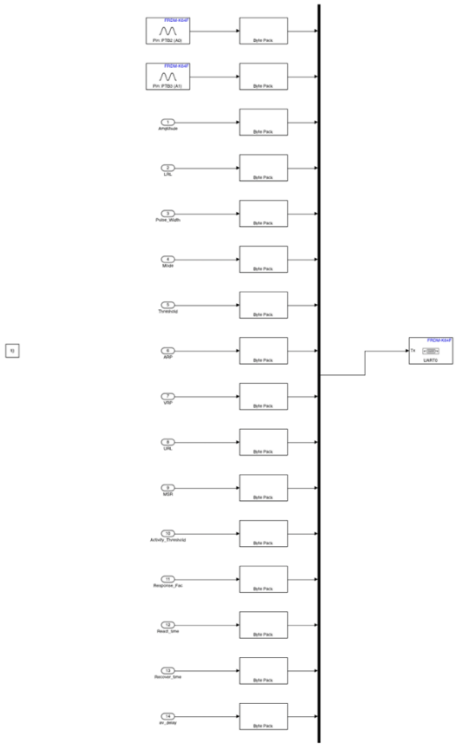
Rate Adaptive state flow



Serial Communication State Flow (Input)



Serial Communication Input Module



Serial Communication Output

Part 2: DCM Design Documentation

DCM Assignment 1

1. Requirement

The Device Controller-Monitor (DCM) module of this system aims to provide an intuitive user interface, enabling users to easily select and configure pacemaker modes (such as AOO, VOO, AAI, and VVI) and related parameters. The DCM module must also ensure the security of user data and provide scalability to accommodate future functional requirements. The following describes each functional, interface, and data management requirement in detail.

1.1 The DCM interface shall support four basic pacemaker modes: AOO, VOO, AAI, and VVI modes.

- **Description**: The interface should provide four pacemaker mode buttons in an intuitive layout, allowing users to switch between AOO, VOO, AAI, and VVI modes. After selecting a mode, the system should provide feedback indicating the selected mode to confirm the validity of the input.
- **Source**: Based on Assignment 1 project requirements and PACEMAKER specifications.

1.2 The DCM interface shall support configuration of multiple pacemaker parameters, including the Lower Rate Limit (LRL), Upper Rate Limit (URL), pulse amplitude, pulse width, and atrial and ventricular refractory periods (ARP, VRP).

- **Description**: Users should be able to enter these parameters in the interface, ensuring that each parameter remains within the designated range (e.g., LRL between 30 and 175 BPM, URL between 50 and 200 BPM) to prevent invalid input. The system shall provide automated input validation and error messaging to ensure data accuracy.
- **Source**: Based on Assignment 1 project requirements, Section 2.1.

2.1 The system shall support the registration and login of at least 10 users and ensure the security of user credentials.

- **Description**: Users should be able to register a unique username and password and log in during subsequent sessions. The system must store user information in a secure structure to prevent unauthorized access, and each user account shall support independent pacemaker parameter settings.
- **Source**: Assignment 1 project requirements, Section 2.2.

2.2 The system shall encrypt user credentials (e.g., passwords) to ensure secure transmission and storage of information.

- **Description**: The system shall employ a standard encryption algorithm (such as bcrypt) to securely store user credentials, preventing unauthorized reading or access. Each time a user inputs credentials, the system shall compare them with encrypted data in storage to confirm the user's identity.
- **Source**: Assignment 1 project requirements, Section 2.3.

3.1 The system shall validate all parameters entered by users to ensure they are within the specified ranges.

- **Description**: The system shall set input ranges for each parameter, such as an LRL range of 30 to 150 BPM, a URL range of 50 to 200 BPM, an amplitude range of 0.5V to 7.0V, and a pulse width range of 0.05ms to 1.9ms. Each time an input falls outside the allowed range, the system shall immediately provide an error message to prevent invalid data from entering the system.
- **Source**: Assignment 1 project requirements, Section 2.2.

4.1 The main interface of the DCM shall provide clear navigation to allow users to switch between mode selection and parameter configuration functions.

- **Description**: The main interface shall include login, registration, and pacemaker mode selection buttons. After successful login, users should be able to select one of the four pacemaker modes and enter the respective parameter configuration window. Each configuration window should be simple and clear, displaying corresponding input fields and unit labels to guide users in entering parameters correctly.
- **Source**: Based on Assignment 1 project requirements, Section 2.2.

4.2 The DCM interface shall provide appropriate feedback information, such as mode confirmation and parameter error messages, to enhance user experience.

- **Description**: After each mode selection, the system shall confirm the selected mode through a popup window or message box, and display error messages when parameters are entered incorrectly. Feedback information should be clearly visible to help users quickly understand the system status.
- **Source**: Assignment 1 project requirements, Section 2.3.

2. Design Decisions

Language and GUI framework: Python was chosen for the development of the DCM due to its ease of use, stability, and available libraries for graphical user interfaces (tkinter) and probable future serial communication (i.e. PySerial). The user interface was designed to be simple and intuitive, with clear input fields for programmable parameters and buttons for selecting pacing modes, so that unpracticed users can learn to use it with minimal effort.

User Data Management: User data (username and password) is stored in a dictionary. This simple data structure is suitable for handling a small number of users.

Input Validation: Constraints were applied to each input field to ensure that parameters like rate limits, amplitude, and pulse width remain within the allowed ranges. Errors are displayed to the user when inputs fall outside of these ranges.

Mode Selection Interface: The pacemaker modes can be selected with the buttons provided by the GUI. Once a mode is selected, the system displays a feedback message to notify the user.

3. Testing and Verification

As required by the assignment 1, the system was tested for the following functions:

User Registration: The system correctly registers new users if the username is unique. If a duplicate username is entered, an error message is displayed.

User Login: Login functionality was tested for both valid and invalid credentials. The system displayed success messages for valid logins and error messages for invalid credentials.

Mode Selection: The mode selection buttons were tested, and the system correctly provided feedback message when a mode was chosen.

3.1. Code description

1) Root home menu

```
119 # Main window and functionality
120 def open_main_window():
121     global main_window  # Declare main_window as a global variable
122     main_window = tk.Tk()
123     main_window.title("DCM Main Interface")
124     main_window.geometry("400x300")
125 
126     # Add menu bar
127     menu_bar = tk.Menu(main_window)
128 
129     # Operations menu
130     operation_menu = tk.Menu(menu_bar, tearoff=0)
131     operation_menu.add_command(label="Device Status", command=show_device_status)
132     operation_menu.add_command(label="Refresh Connection",
    command=refresh_connection_status)
133     operation_menu.add_separator()
134     operation_menu.add_command(label="Exit", command=main_window.quit)
135     menu_bar.add_cascade(label="Operations", menu=operation_menu)
```

By using python's standard Gui interface, we can make a Ui based on the default window parameters of the public version, which contains four different functional interfaces, as the main interface after login, provide a jump platform, and automatically close after selecting the interface of the next directory. Users can select AOO, VOO, AAI, VVI and other pacemaker modes, and open the parameter input window of each mode, allowing users to select specific pacemaker modes after successful login.

2）Input Validation Function

```python
def validate_input(LRL, URL, amplitude, pulse_width):
    try:
        LRL = round(float(LRL), 5)
        URL = round(float(URL), 5)
        amplitude = round(float(amplitude), 5)
        pulse_width = round(float(pulse_width), 5)

        if not (30 <= LRL <= 150):
            raise ValueError("Lower Rate Limit must be between 30 and 150.")
        if not (50 <= URL <= 180):
            raise ValueError("Upper Rate Limit must be between 50 and 180.")
        if not (0.5 <= amplitude <= 5.0):
            raise ValueError("Amplitude must be between 0.5V and 5.0V.")
        if not (0.05 <= pulse_width <= 1.9):
            raise ValueError("Pulse Width must be between 0.05ms and 1.9ms.")
        if (URL <= LRL):
            raise ValueError("Warning: Lower Rate cannot exceed Upper Rate.")
        return True

    except ValueError as e:
        messagebox.showerror("Input Error", str(e))
        return False
```

This section verifies the input of various parameters such as LRL (lower limit rate limit), URL (upper limit rate limit), amplitude, and pulse width by using logic gates.

LRL must be between 30 and 150.

The URL must be between 50 and 180.

The amplitude must be between 0.5V and 5.0V.

Pulse width must be between 0.05ms and 1.9ms.

This is to ensure that only valid data is accepted before it is sent to the serial interface.

3） User registration process

```
81      def register():
82          username = entry_username.get()
83          password = entry_password.get()
84          code = entry_invite_code.get()
85
86          if len(users) >= 10:
87              messagebox.showerror("ERROR", "User limit reached. Cannot register more
    users.")
88              register_window.destroy()
89              return
90          if code != invite_code:
91              messagebox.showerror("ERROR", "Invalid invite code.")
92              return
93          if username in users:
94              messagebox.showerror("ERROR", "Username already exists.")
95              return
96          if len(password) < 6:
97              messagebox.showerror("ERROR", "Password must be at least 6 characters
    long.")
98              return
```

This part is mainly to provide the user group and user information management process. In this block, since we previously registered new users in the program and stored all user information on the heap, only variable transformation was carried out in the program. Therefore, in the process of changing the program structure, we added transfer function to store the username and password to a specific file name as an addressing entry. This is used to store passwords.

4）Data storage process

```
248 # Save parameters to file
249 def save_parameters_to_file(mode, parameters):
250     filename = f"{mode}_parameters.txt"
251     with open(filename, 'w') as file:
252         for key, value in parameters.items():
253             file.write(f"{key}: {value}\n")
254     messagebox.showinfo("SUCCESS", f"Parameters saved to {filename}")
255
```

This part is the part of the code dealing with the data storage process, which mainly includes the user input information storage, encryption(still in TIB), and the process of comparing with the encrypted information. To ensure the security of personal

information, the information does not go through the heap after input, goes through the encrypted part, and is compared with the database (that is, the encrypted information in the name of the personal profile) to determine whether the user has registered, or whether the user password is correct.

5） User login

```
108 # Login functionality
109 def login():
110     username = entry_username.get()
111     password = entry_password.get().encode('utf-8')
112 
113     if username in users and bcrypt.checkpw(password,
    users[username].encode('utf-8')):
114         messagebox.showinfo("SUCCESS", f"Login successful! User: {username}")
115         open_main_window()
116     else:
117         messagebox.showerror("ERROR", "Incorrect username or password!")
118 
```

```
49 # Load user data file
50 def load_users():
51     global users
52     if os.path.exists(user_file):
53         with open(user_file, 'r') as file:
54             users = json.load(file)
55     else:
56         users = {}
57 
58 # Save user data
59 def save_users():
60     with open(user_file, 'w') as file:
61         json.dump(users, file)
62 
```

This is the primary operation procedure of this program, mainly for the primary acquisition and detection of personal information, when the program is running, the input information is first obtained into the stack, and then the compliance of the data is

checked, and then compared with the existing user information in the database to determine whether it is repeated registration.

6） 3 major functional blocks

```
def open_VVI():
    VVI_window = tk.Tk()
    VVI_window.title("VVI Parameters")
    label_VVI = tk.Label(VVI_window, text="
    label_VVI.pack(pady=20)
    label_LRL = tk.Label(VVI_window, text="Lower Rate Limit").pack(pady=5)
    entry_LRL = tk.Entry(VVI_window)
    entry_LRL.pack(pady=5)
    label_URL = tk.Label(VVI_window, text="Upper Rate Limit").pack(pady=5)
    entry_URL = tk.Entry(VVI_window)
    entry_URL.pack(pady=5)
    label_VA = tk.Label(VVI_window, text="Ventricular Amplitude").pack(pady=5)
    entry_VA = tk.Entry(VVI_window)
    entry_VA.pack(pady=5)
    label_VPW = tk.Label(VVI_window, text="Ventricular Pulse Width").pack(pady=5)
    entry_VPW = tk.Entry(VVI_window)
    entry_VPW.pack(pady=5)
    label_VRP = tk.Label(VVI_window, text="VRP").pack(pady=5)
    entry_VRP = tk.Entry(VVI_window)
    entry_VRP.pack(pady=5)


    # Displays the results after enter button is clicked
    def VVI_results():
        LRL = entry_LRL.get()
        URL = entry_URL.get()
        amplitude = entry_VA.get()
        pulse_width = entry_VPW.get()
        VRP = entry_VRP.get()
```

```
def open_AAI():
    AAI_window = tk.Tk()
    AAI_window.title("AAI Parameters")
    label_AAI = tk.Label(AAI_window, text="                                                    Choose your AAI parameters.
    label_AAI.pack(pady=20)
    label_LRL = tk.Label(AAI_window, text="Lower Rate Limit").pack(pady=5)
    entry_LRL = tk.Entry(AAI_window)
    entry_LRL.pack(pady=5)
    label_URL = tk.Label(AAI_window, text="Upper Rate Limit").pack(pady=5)
    entry_URL = tk.Entry(AAI_window)
    entry_URL.pack(pady=5)
    label_AA = tk.Label(AAI_window, text="Atrial Amplitude").pack(pady=5)
    entry_AA = tk.Entry(AAI_window)
    entry_AA.pack(pady=5)
    label_APW = tk.Label(AAI_window, text="Atrial Pulse Width").pack(pady=5)
    entry_APW = tk.Entry(AAI_window)
    entry_APW.pack(pady=5)
    label_ARP = tk.Label(AAI_window, text="ARP").pack(pady=5)
    entry_ARP = tk.Entry(AAI_window)
    entry_ARP.pack(pady=5)


    # Displays the results after enter button is clicked
    def AAI_results():
        LRL = entry_LRL.get()
        URL = entry_URL.get()
        amplitude = entry_AA.get()
        pulse_width = entry_APW.get()
        ARP = entry_ARP.get()
```

```
def open_AOO():
    AOO_window = tk.Tk()
    AOO_window.title("AOO Parameters")
    label_AOO = tk.Label(AOO_window, text="                                                    Choose your AOO parameters.
    label_AOO.pack(pady=20)

    label_LRL = tk.Label(AOO_window, text="Lower Rate Limit").pack(pady=5)
    entry_LRL = tk.Entry(AOO_window)
    entry_LRL.pack(pady=5)

    label_URL = tk.Label(AOO_window, text="Upper Rate Limit").pack(pady=5)
    entry_URL = tk.Entry(AOO_window)
    entry_URL.pack(pady=5)

    label_AA = tk.Label(AOO_window, text="Atrial Amplitude").pack(pady=5)
    entry_AA = tk.Entry(AOO_window)
    entry_AA.pack(pady=5)

    label_APW = tk.Label(AOO_window, text="Atrial Pulse Width").pack(pady=5)
    entry_APW = tk.Entry(AOO_window)
    entry_APW.pack(pady=5)

    # Displays the results after enter button is clicked
    def AOO_results():
        LRL = entry_LRL.get()
        URL = entry_URL.get()
        amplitude = entry_AA.get()
        pulse_width = entry_APW.get()
```

The main functional block will be required for the pacemaker variables to be transferred. Each module has a module for input information, judgment of the validity of input values, storage of input information to files, and feedback of input information to the user in the form of a new window.

7）Misc

```
3 # Initial data and settings
4 users = {}
5 user_file = "users.json"
6 invite_code = "123456"
7
```

The default password, which is like the Windows administration account, is used to debug programs

```
1 import tkinter as tk
2 from tkinter import messagebox
3 from PIL import Image, ImageTk
4 import bcrypt
5 import json
6 import os
```

Shorthand for python runtime, used to omit complex library writing while faster debugging.

```
# Program entry point
load_users()
login_window = tk.Tk()
login_window.title("DCM System Login")

label_username = tk.Label(login_window, text="Username:")
label_username.pack(pady=5)
entry_username = tk.Entry(login_window)
entry_username.pack(pady=5)

label_password = tk.Label(login_window, text="Password:")
label_password.pack(pady=5)
entry_password = tk.Entry(login_window, show="*")
entry_password.pack(pady=5)

btn_register = tk.Button(login_window, text="Register", command=open_register_window)
btn_register.pack(pady=5)

btn_login = tk.Button(login_window, text="Login", command=login)
btn_login.pack(pady=5)

login_window.mainloop()
```

The program execution queue arranges the execution order of each module according to the execution time sequence and considers the time spent by the time-sharing modules added in each module.

```
try:
    import bcrypt #checking bcrypt situation

except ImportError:

    # Notify the user that bcrypt is missing
    def notify_missing_bcrypt():
        messagebox.showerror(
            "Security Module Not Found",
            "The Python Library 'bcrypt' is not detected.\nPlease install 'bcrypt' to run this program.\n\n\nFor Windows system, use the command in the Windows PowerShell:\n\npip install bcrypt"
        )

    # GUI execution
    notify_missing_bcrypt()
    # Backup SystemExit
    raise SystemExit("bcrypt is required to run this program. Please install it using 'pip install bcrypt' for windows system.")

```

Detection for cypitfix lib, use for security.

```
# Mode selection window
def open_mode_selection():
    mode_window = tk.Toplevel(main_window)
    mode_window.title("Mode Selection")

    btn_AOO = tk.Button(mode_window, text="AOO Mode", command=lambda: open_AOO())
    btn_AOO.pack(pady=5)
    btn_VOO = tk.Button(mode_window, text="VOO Mode", command=lambda: open_VOO())
    btn_VOO.pack(pady=5)
    btn_AAI = tk.Button(mode_window, text="AAI Mode", command=lambda: open_AAI())
    btn_AAI.pack(pady=5)
    btn_VVI = tk.Button(mode_window, text="VVI Mode", command=lambda: open_VVI())
    btn_VVI.pack(pady=5)

```

Easy top FSM design, using for thread limitation, since the Pages jumps out once alone, each of function, or in other word, directory function has to be limited to one, in order to save graphics resources and make the contact process more clear.

```
150 # Device status display
151 def show_device_status():
152     messagebox.showinfo("Device Status", "Device Connected: Yes\nPort: N/A\nBattery: 75%")
153
154 # Refresh connection status
155 def refresh_connection_status():
156     messagebox.showinfo("Connection Status", "Connection Refreshed.\nStatus: Connected")
157
158 # ECG display
159 def show_ecg_graph():
160     ecg_window = tk.Toplevel(main_window)
161     ecg_window.title("ECG Display")
162     label = tk.Label(ecg_window, text="ECG Graph will be displayed here.")
163     label.pack(pady=20)
164
```

These three functions are still in the stage of development, which is the feedback from the device which connected with serial, handling I/O, basic info of the device including PID control, dashboard and system monitoring and so on.

2.1. Likely Requirement Changes

Future requirement changes to the DCM system may include:

Accessibility: Enhancements for users with disabilities, such as support for visually impaired users, colorblind-friendly settings, and text-to-speech features, as well as supporting multiple languages.

Telemetry Support: Integration with the pacemaker for real-time serial communication, ultimately completing the 3K04 Pacemaker project.

Additional Modes: Expansion of the mode selection functionality to include more pacemaker modes (such as AOOR, VOOR, AAIR, VVIR) in the code. This can be done with additional classes in the code.

Emergency Stop: add functions to terminate the pacemaker if necessary.

Session Management: Implement session management functionality to maintain user login states and preferences across multiple interactions within the app. This will ensure that users do not need to re-authenticate during their session, providing a seamless experience as they navigate through different features.

2.2. Likely Design Changes

Future design changes to the DCM system may include:

Visual changes: Enhance the visual appeal of the app by adding colors and animations to make it more engaging for users.

App clarity: Adding intuitive navigation buttons in the top bar, with the electrogram displayed in the center of the screen. Maybe add tooltips or a tutorial to help guide the user on using the system. By clearly defining each feature and making it easily accessible, users will be able to navigate the app effortlessly and understand its functions without confusion.

Cross compatible on devices: Ensure the system can be used for various devices and screen sizes and provide a consistent user experience across platforms.

Feedback Mechanisms: Include a feedback button to gather insights from users and improve the app further based on their experience.

Rounding Input: add functions to round input parameters to an appropriate increment.

2.3. Module Descriptions

1. User Management Module

(a) **Purpose**: This module handles user registration and login functionality. It ensures that users can securely log in to the system or register new accounts.

(b) **Public Functions:**

- register (): Registers a new user. Takes the username and password from the input fields, checks if the username already exists, and either registers the user or displays an error.
    - **Parameters**: None (uses global input fields entry_username and entry_password).
- login (): Authenticates an existing user by checking the username and password. If valid, the user is logged in, and the main window is opened. If invalid, an error message is displayed.
    - **Parameters**: None (uses global input fields entry_username and entry_password).

(c) **Black-Box Behavior**:

- register (): When called, checks if the username already exists in the users dictionary. If it does not, the function adds the new user and displays a success message.
- login (): Compares the entered credentials with those stored in the users dictionary. If they match, a success message is displayed, and the main window opens. If they do not match, an error is shown.

(d) **State Variables**:

- users: A dictionary containing username-password pairs. This dictionary is shared across the registration and login functions to ensure consistent access to user data.

(e) **Private Functions**: None.

(f) **Internal Behavior**:

- The register () function retrieves the values from the username and password fields, checks if the username already exists, and updates the user's dictionary. If the username exists, an error message is shown; otherwise, the new user is registered.
- The login () function compares the entered username and password with the entries in the user's dictionary. If the credentials match, the login is successful; otherwise, an error is displayed.

2. Main Interface Module

(a) **Purpose**: This module manages the main interface of the DCM system, allowing users to select pacemaker modes and receive feedback on their choices.

(b) **Public Functions**:

- open_main_window(): Opens the main DCM interface, providing options for selecting pacemaker modes.
    - **Parameters**: None.
- select_mode(mode): Displays a message indicating the selected pacemaker mode.

- **Parameters**: mode (string) – the name of the selected mode (AOO, VOO, AAI, VVI).

(c) **Black-Box Behavior**:

- open_main_window(): This function closes the login window and opens the main interface, where the user can select a pacemaker mode.
- select_mode(): Displays a message box indicating the chosen pacemaker mode.

(d) **State Variables**: None. The function open_main_window() destroys the login window and opens a new window for the main interface.

(e) **Private Functions**: None.

(f) **Internal Behavior**:

- The open_main_window() function first destroys the login window to prevent multiple windows from being open simultaneously. It then creates the main interface, which includes buttons for mode selection.
- The select_mode(mode) function takes the selected mode as an argument and displays a feedback message to the user, confirming the selected mode.

2.3. Testing and Results

As mentioned earlier, the DCM system was tested for:

1. **User Registration Testing**:
    - **Test Case**: Registering a new user with a unique username.

        - **Expected Outcome**: The system successfully registers the user and displays a success message.
        - **Result**: Pass.
    - **Test Case**: Registering a new user with an existing username.
        - **Expected Outcome**: The system displays an error message stating that the username already exists.
        - **Result**: Pass.

2. **User Login Testing**:
    - **Test Case**: Logging in with correct credentials.
        - **Expected Outcome**: The system successfully logs in the user and opens the main interface.
        - **Result**: Pass.
    - **Test Case**: Logging in with incorrect credentials.
        - **Expected Outcome**: The system displays an error message stating that the username or password is incorrect.
        - **Result**: Pass.

3. **Mode Selection Testing**:
    - **Test Case**: Selecting any pacemaker mode (AOO, VOO, AAI, VVI).
        - **Expected Outcome**: A message box displays the selected mode.

- **Result**: Pass.

DCM Assignment 2

1.Requirements

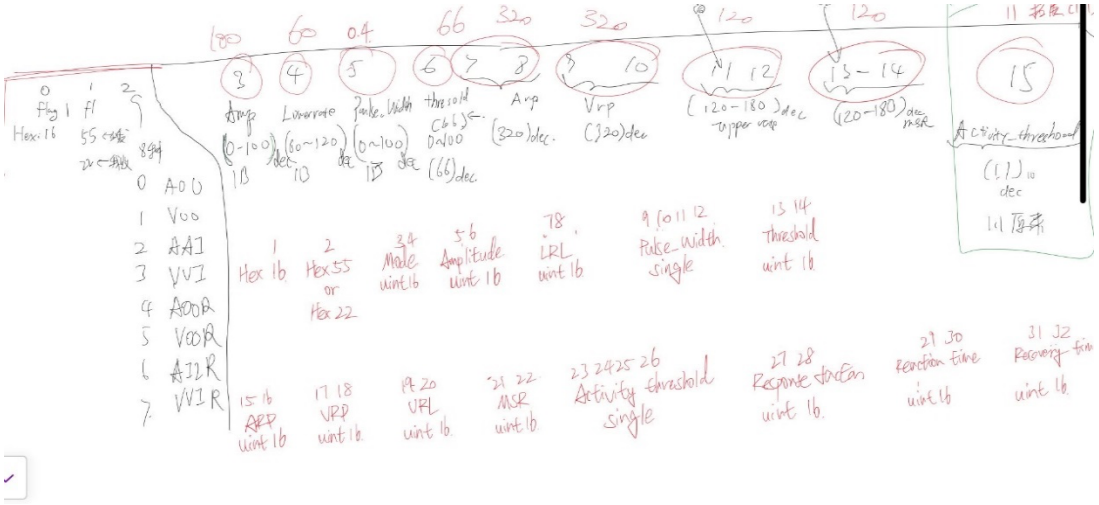
The Device Controller-Monitor (DCM) is designed as an interface to manage pacemaker operations. It supports eight pacing modes such as AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR. The DCM must facilitate data exchange through serial communication, generate electrogram graphs, and provide control functionalities. The DCM must also include safety features, such as verifying that the received data matches the sent data. These features enable real-time interaction with pacemaker hardware, ensuring a smooth and intuitive user experience.

Key implementation of DCM system:

Serial-Uart bitstream:

Description	Size	Bit director in bitstream
Handshaking	Int 8	0:1 -> determined
UART->	Int 8	2:3 -> 0x55 is pure sending 0x22 is receiving
Mode	Uint8	3:4 -> 0—7 is from AOO to VVIR
Amplitude	uint16(2 bytes)	5:6
LRL (Lower rate limit)	uint16(2 bytes)	7:8
Pulse_Width	single(4 bytes)	9:12

Threshold	uint16(2 bytes)	13:14
ARP	single(4 bytes)	15:18
VRP	single(4 bytes)	19:22
URL(Upper rate limit)	uint16(2 bytes)	23:24
MSR	uint16(2 bytes)	25:26
Activity_Threshold	single(4 bytes)	27:30
Response_factor	uint16(2 bytes)	31:32
Reaction_time	uint16(2 bytes)	33:34
Recovery_time	uint16(2 bytes)	35:36



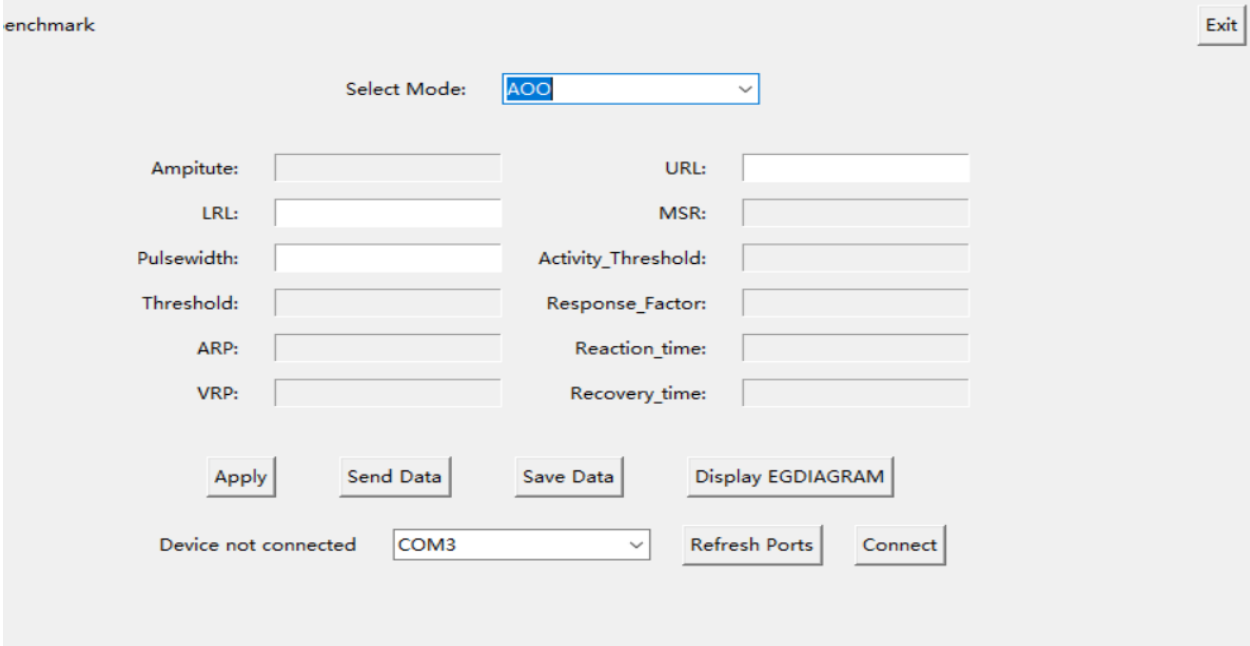
Serial to the PPmaker:

```python
import serial
import struct
import time
from tkinter import messagebox
from serial.tools import list_ports  # 用于列出可用的串口

class SerialManager:
    def __init__(self, port="COM3", baudrate=115200, timeout=1):
        self.port = port
        self.baudrate = baudrate
        self.timeout = timeout
        self.serial_port = None

    def list_available_ports(self):
        """列出所有可用的串口"""
        ports = list_ports.comports()  # 获取所有可用串口
        available_ports = [port.device for port in ports]
        return available_ports

    def connect(self):
        """连接到串口设备"""
        try:
            self.serial_port = serial.Serial(self.port, self.baudrate, timeout=self.timeout)
            if self.serial_port.is_open:
                return True
        except serial.SerialException as e:
            print(f"无法连接到 {self.port}: {e}")
            return False

    def disconnect(self):
        """断开串口连接"""
        if self.serial_port and self.serial_port.is_open:
```

EGdiagram:

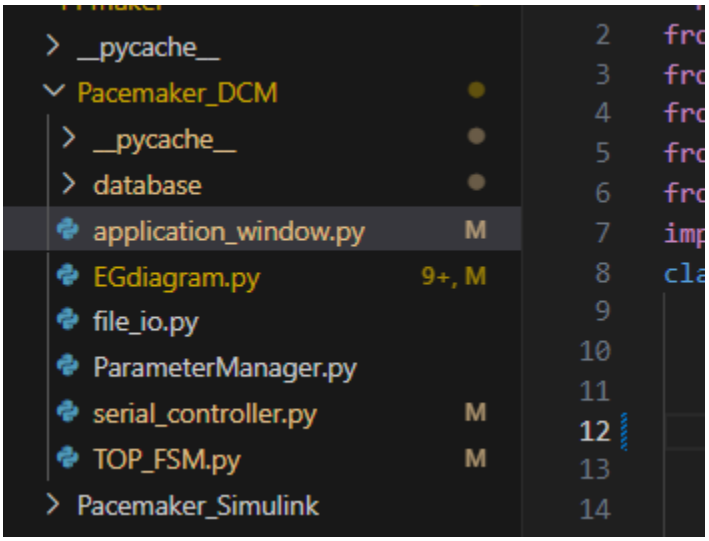
```python
import time

# 线程函数，用于在DCM窗口上绘制ECG图
class rander:
    def __init__(self):
        self.__displayGraph
        self.__annotations__

    def __displayGraph(self):
        """
        线程函数，用于绘制ECG图
        """
        global write
        t = time.time()  # 记录当前时间
        tvlist = []  # 存储心电图的时间戳
        talist = []  # 存储心电图的幅度
        voltageV = []  # 存储电压V数据
        voltageA = []  # 存储电压A数据
        lasttime = t  # 最后更新时间

        write = True  # 初始为未写入状态
        print(write)
        print(sc.getCurrentPort())
        print(sc.serialWrite(b'\x16\x00\x22'))
        while not write:
            # 判断串口是否打开
            if not (sc.getCurrentPort() is None):
                if not write:
                    # 发送数据到串口
                    sc.serialWrite(b'\x16\x00\x22')
                    print(sc.serialWrite(b'\x16\x00\x22'))
                    temp = sc.serial_read()  # 读取串口数据
```

UI updates:

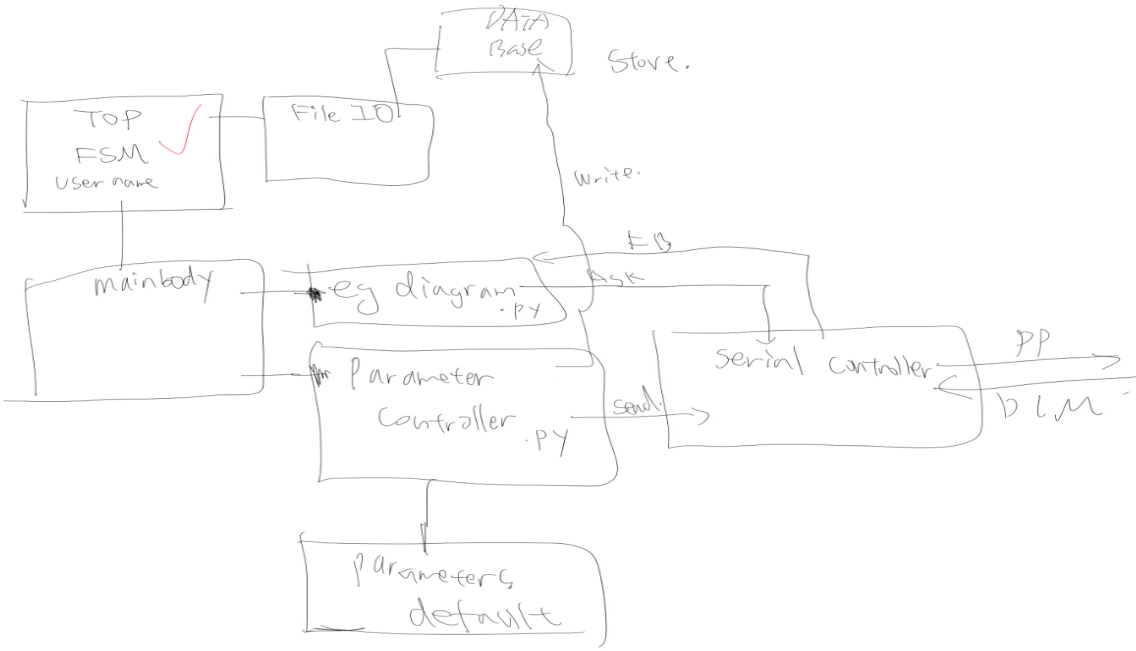


Front end and backend stability enhancements

OOD and module design



The DCM architecture:



TOP_FSM.py

Overview

The `TOP_FSM.py` file is where the program starts. It allows users to either register a new account or log in to their existing account. After a successful login, users are redirected to the main application window to access additional functionalities.

Constructor: __init__

Here's the code for this function:

```python
import tkinter as tk
from tkinter import messagebox
import os
from file_io import FileIO
from application_window import ApplicationWindow

class MainWindow:
    def __init__(self, root):
        # 设置主窗口
        self.root = root
        self.root.title("This is a Eemergency SOFTWARE")
        self.root.geometry("600x360")

        # 初始化 FileIO
        self.file_io = FileIO(os.path.dirname(os.path.abspath(__file__)))

        # 欢迎信息
        mainline = tk.Label(root, text="Quartz Extreme PPmaker")
        mainline.pack(pady=20)

        # 用户名输入框
        self.usrname = tk.Entry(root)
        self.usrname.insert(0, "Username")
        self.usrname.pack(pady=10)

        # 密码输入框
        self.password_input = tk.Entry(root)
        self.password_input.insert(0, "Password")
        self.password_input.pack(pady=10)

        # 按钮区域
        btn_frame = tk.Frame(root)
        btn_frame.pack(pady=15)

```

```
        # 注册按钮
        register_btn = tk.Button(btn_frame, text="signed", command=self.register_user, width=15)
        register_btn.pack(side=tk.LEFT, padx=5)

        # 登录按钮
        login_btn = tk.Button(btn_frame, text="login", command=self.login_user, width=15)
        login_btn.pack(side=tk.LEFT, padx=5)

        # 退出按钮
        exit_btn = tk.Button(root, text="quit", command=self.root.quit, width=5)
        exit_btn.pack(side=tk.RIGHT, padx=20)

        # 成功信息标签
        self.success_msg = tk.Label(root, text="ACT")
        self.success_msg.pack(pady=20)
```

This function is responsible for setting up the main login window. The `root` parameter is the main Tkinter window passed to the class. Inside the constructor, the program creates the user interface components, including text boxes for username and password, buttons for login and registration, and a label to display messages like 'login success' or errors.

The username and password fields are created using the `tk.Entry` widget, with default placeholders ('Username' and 'Password'). The buttons are added to a separate frame (`btn_frame`) for a neat layout. Each button has a specific command attached, such as `self.register_user` or `self.login_user`, which are functions that handle their respective actions.

register_user (Sign Up)

```
51      def register_user(self):
52          """
53          注册用户的函数
54          """
55          username = self.usrname.get().strip()
56          password = self.password_input.get().strip()
57
58          if username and password:
59              success = self.file_io.save_user(username, password)
60              if success:
61                  self.success_msg.config(text="sign success")
62                  self.usrname.delete(0, tk.END)
63                  self.password_input.delete(0, tk.END)
64              else:
65                  messagebox.showwarning("error, user has already in the database")
66          else:
67              messagebox.showwarning("error please input username or password")
68
```

This function handles the user registration process. It first retrieves the username and password entered by the user using `self.usrname.get()` and `self.password_input.get()`. It then trims any extra spaces using `.strip()`. If both fields are filled, the program calls `self.file_io.save_user` to save the account information.

If the username is already in the database, a warning message appears using `messagebox.showwarning`. On successful registration, it clears the input fields and updates the success message to 'sign success'. If either field is empty, the program prompts the user to fill in the missing information.

login_user (Log In)

```
69      def login_user(self):
70          """
71          用户登录的函数
72          """
73          username = self.usrname.get().strip()
74          password = self.password_input.get().strip()
75
76          if username and password:
77              user_data = self.file_io.load_user(username)
78              if not user_data:
79                  messagebox.showwarning("error", "no user account")
80                  return
81
82              if user_data.get('password') == password:
83                  self.success_msg.config(text="login in success")
84                  self.open_application_window(username)
85              else:
86                  messagebox.showwarning("error no passwordW")
87          else:
88              messagebox.showwarning("error", "please input password")
89
```

This function manages user login. It works similarly to `register_user`, but instead of saving data, it retrieves stored information using `self.file_io.load_user`. The entered username and password are compared with the stored data.

If the username doesn't exist in the database, a warning appears. If the password matches, the success message updates to 'login in success', and the program calls `self.open_application_window` to open the main application. If the password is incorrect, it displays an error message.

open_application_window (Open Main App)

```
90      def open_application_window(self, username):
91          """
92          打开应用程序窗口
93          """
94          app_window = tk.Toplevel(self.root)
95          ApplicationWindow(app_window, username)
96          self.root.withdraw()
97
98  def main():
```

Main Function

```
98  def main():
99      """
100     主程序入口
101     """
102     root = tk.Tk()
103     MainWindow(root)
104     root.mainloop()
105
106 if __name__ == "__main__":
107     main()
108
```

The `main` function starts the entire program. It creates the root Tkinter window and initializes the `MainWindow` class. The `root.mainloop()` method ensures the window stays open, waiting for user interaction.

Detailed Explanation of serial_controller.py

Overview

The `serial_controller.py` file handles all serial communication tasks. It manages connecting to the device, sending and receiving data, and building data packets to ensure smooth interaction between the software and the hardware.

Main Class: SerialManager

Constructor: __init__

Here's the code for this function:

```python
import serial
import struct
import time
from tkinter import messagebox
from serial.tools import list_ports  # 用于列出可用的串口

class SerialManager:
    def __init__(self, port="COM3", baudrate=115200, timeout=1):
        self.port = port
        self.baudrate = baudrate
        self.timeout = timeout
        self.serial_port = None
```

The constructor sets up the initial parameters for the serial connection, such as the port, baud rate, and timeout duration. The `serial_port` variable is initialized to `None`, indicating no connection is active yet. These parameters can be adjusted as needed depending on the target device.

Function: list_available_ports (List Ports)

Here's the code for this function:

```
def list_available_ports(self):
    """列出所有可用的串口"""
    ports = list_ports.comports()  # 获取所有可用串口
    available_ports = [port.device for port in ports]
    return available_ports
```

This function lists all available COM ports on the system using the `list_ports.comports()` method. It creates a list of port names (e.g., COM3, COM4) and returns it. This is helpful for users to select the correct port for their device.

Function: connect (Connect to Device)

```
def connect(self):
    """连接到串口设备"""
    try:
        self.serial_port = serial.Serial(self.port, self.baudrate, timeout=self.timeout)
        if self.serial_port.is_open:
            return True
    except serial.SerialException as e:
        print(f"无法连接到 {self.port}: {e}")
        return False
```

This function attempts to connect to the serial device using the port, baud rate, and timeout defined earlier. It creates a `Serial` object and checks if the connection is open. If successful, it returns `True`. Otherwise, it prints an error message and returns `False`. This notifies the user of connection issues.

Function: disconnect (Disconnect from Device)

Here's the code for this function:

```
def disconnect(self):
    """断开串口连接"""
    if self.serial_port and self.serial_port.is_open:
        self.serial_port.close()
```

This function safely closes the connection to the device if it is currently open. It ensures that resources are properly released after the communication is complete.

Function: is_connected (Check Connection)

Here's the code for this function:

```
def is_connected(self):
    """检查串口是否连接"""
    return self.serial_port and self.serial_port.is_open
```

This function checks if the serial port is currently connected and open. It returns `True` if connected and `False` otherwise. This is useful for validating the connection status before sending data.

Function: send_data (Send Data)

```
39      def send_data(self, data):
40          """发送数据到串口设备"""
41          if self.is_connected():
42              try:
43                  self.serial_port.write(data)
44                  print(data)
45                  return True
46              except Exception as e:
47                  print(f"发送数据时出错: {e}")
48                  return False
49          else:
50              print("设备未连接")
51              return False
52
```

This function sends data to the connected device through the serial port. It first checks if the device is connected using the `is_connected` function. If connected, it writes the data to the port and prints it for debugging purposes. If there's an error, it returns `False` and prints an error message.

build_data_packet (Build Data Packet)

```
53      def build_data_packet(self, field_values):
54          """根据字段值构建数据包"""
55          try:
56              header = struct.pack('<B', 0x16)  # 示例头部
57              body = struct.pack('<B', 0x55)  # 示例头部
58              mod = struct.pack('<2B', 0x00)  # 示例头部
59              data_format = '<2B2B2B4f2B4B4B2B2B2B4f2B2B2B2B'  # 示例数据格式
60              data = struct.pack(
61                  data_format,
62                  int(field_values["Ampitute"]),
63                  int(field_values["LRL"]),
64                  float(field_values["Pulsewidth"]),
65                  int(field_values["Threshold"]),
66                  float(field_values["ARP"]),
67                  float(field_values["VRP"]),
68                  int(field_values["URL"]),
69                  int(field_values["MSR"]),
70                  float(field_values["Activity_Threshold"]),
71                  int(field_values["Response_Factor"]),
72                  int(field_values["Reaction_time"]),
73                  int(field_values["Recovery_time"]),
74              )
75              return header + body + mod + data
76          except Exception as e:
77              print(f"构建数据包时出错: {e}")
78              return None
79
```

This function creates a data packet for sending to the device. It combines multiple fields into a structured binary format using `struct.pack`. The packet includes a header, body, and additional data based on the provided `field_values`. If there's an error, it prints an error message and returns `None`. This ensures the data is sent in the correct format the device expects.

Detailed Explanation of ParameterManager.py

Overview

The `ParameterManager.py` file is a key component in the application. It provides a structured way to manage the various parameters related to the system. This class initializes default values for parameters, provides methods to retrieve these values, and allows for updating them with proper validation.

Class: ParameterManager

Constructor: __init__

Code:

```
class ParameterManager:
    def __init__(self):
        #default val
        self.__amplitude = 100
        self.__lrl = 60
        self.__pulse_width = 0.4
        self.__threshold = 66
        self.__arp = 320
        self.__vrp = 320
        self.__url = 120
        self.__msr = 120
        self.__activity_threshold = 1.1 # Med
        self.__response_factor = 8
        self.__reaction_time = 10
        self.__recovery_time = 30  # sec = 5 min
```

The constructor initializes default values for all parameters. These include settings like amplitude, lower rate limit (LRL), and pulse width, which are critical to the system's functioning. Each parameter is given a private attribute name (e.g., `__amplitude`) to encapsulate the values and ensure they can only be accessed through defined methods.

Getter Methods

```
def getAmplitude(self):
    return self.__amplitude if self.__amplitude != 0 else 0

def getLowerRateLimit(self):
    return self.__lrl

def getPulseWidth(self):
    return self.__pulse_width

def getthresholf(self):
    return self.__threshold

def getARP(self):
    return self.__arp
def getVRP(self):
    return self.__vrp
```

```
29      def getARP(self):
30          return self.__arp
31      def getVRP(self):
32          return self.__vrp
33      def getUpperRateLimit(self):
34          return self.__url
35      def getMSR(self):
36          return self.__msr
37      def getActivityThreshold(self):
38          if self.__activity_threshold - 1.13 < 0.01:
39              return
40          elif self.__activity_threshold - 1.25 < 0.01:
41              return 'L'
42          elif self.__activity_threshold - 1.4 < 0.01:
43              return 'M-L'
44          elif self.__activity_threshold - 1.6 < 0.01:
45              return 'M'
46          elif self.__activity_threshold == 2:
47              return 'M-H'
48          elif self.__activity_threshold - 2.4 < 0.01:
49              return 'H'
50          elif self.__activity_threshold == 3:
51              return 'V-H'
52          return 0
53      def getResponseFactor(self):
54          return self.__response_factor
55      def getReactionTime(self):
56          return self.__reaction_time
57      def getRecoveryTime(self):
58          return round(self.__recovery_time / 60)
```

(variable) __activity_threshold: float

This method returns the amplitude value. If the amplitude is set to zero, it ensures the return value is also zero. This approach helps maintain system integrity when parameters like amplitude must meet specific conditions.

Each getter follows a similar pattern. For example, `getLowerRateLimit` retrieves the current value of the lower rate limit (LRL), while methods like `getActivityThreshold` interpret numerical thresholds and return a human-readable description (e.g., 'Medium', 'High').

Setter Methods

```
#set amp
def setAmplitude(self, val):
    if str(val).casefold() == 'off'.casefold():
        self.__amplitude = 0
        return
    if self.__is_num(val):
        num = round(float(val), 1)
        if num <= 5.0 and num >= 0.1:
            self.__amplitude = num
        elif round(float(val), 1) == 0:
            self.__amplitude = 0
        else:
            raise IndexError
    else:
        raise TypeError
```

This method updates the amplitude parameter. It first checks if the input value is numeric using the helper method `__is_num`. Then, it ensures the value falls within the allowed range (0.1 to 5.0). If not, it raises an error to prevent invalid settings.

Helper Method: __is_num

```
183     def __is_num(self, s):
184         try:
185             float(s)
186         except ValueError:
187             return False
188         else:
189             return True
190
```

This private method checks if a value can be converted to a float. It is used throughout the class to ensure only valid numeric inputs are processed.

Other setter methods follow similar validation logic. For instance, `setLowerRateLimit` ensures the LRL is within the range of 30 to 175 and is a multiple of 5. If the input does not meet these conditions, an error is raised.

```
76      #set lrl
77      def setLowerRateLimit(self, val):
78          if self.__is_num(val):
79              num = 5 * round(float(val) / 5)
80              if round(float(val)) <= 90 and round(float(val)) >= 50:
81                  self.__lrl = round(float(val))
82              elif (num <= 50 and num >= 30) or (num <= 175 and num >= 90):
83                  self.__lrl = num
84              else:
85                  raise IndexError
86          else:
87              raise TypeError
```

This method ensures the value for LRL adheres to strict rules, such as being in predefined ranges and rounded to multiples of 5.

Detailed Explanation of file_io.py

Overview

The `file_io.py` file is responsible for handling file operations, such as saving user credentials, loading user data, and managing user-specific parameters. It ensures all user-related information is stored persistently and securely.

Method Explanations

__init__ Method

```python
import os

class FileIO:
    def __init__(self, base_path):
        self.base_path = base_path
        self.db_folder = os.path.join(self.base_path, "database")

        # 如果数据库文件夹不存在，则创建
        if not os.path.exists(self.db_folder):
            os.makedirs(self.db_folder)

```

This method initializes the `FileIO` class. It sets up the base path for file operations and ensures a 'database' folder exists. If the folder is missing, it creates it to store user-related files.

save_user Method

```
12      def save_user(self, username, password):
13          """
14          保存用户名和密码到文件
15          """
16          user_file_path = os.path.join(self.db_folder, f"{username}.txt")
17
18          if os.path.exists(user_file_path):
19              return False  # 用户名已存在
20
21          with open(user_file_path, 'w') as file:
22              file.write(f"Password: {password}\n")
23          return True
```

This method saves a new user's credentials. It creates a file named after the username in the 'database' folder. If the file already exists (indicating the username is taken), it returns `False`. Otherwise, it writes the password to the file in the format `Password: password` and returns `True`.

load_user Method

```
25      def load_user(self, username):
26          """
27          从文件加载用户信息
28          """
29          user_file_path = os.path.join(self.db_folder, f"{username}.txt")
30
31          if not os.path.exists(user_file_path):
32              return None  # 用户名不存在
33
34          with open(user_file_path, 'r') as file:
35              lines = file.readlines()
36
37          user_data = {}
38          for line in lines:
39              if line.startswith("Password:"):
40                  user_data['password'] = line.split(":")[1].strip()
41
42          return user_data
```

This method retrieves user data from the file corresponding to the given username. It first checks if the file exists. If it does, the file is read line by line, and the password is extracted and stored in a dictionary. If the file doesn't exist, it returns `None`.

write_parameter Method

```python
    def write_parameter(self, values, username):
        """
        将参数值写入与用户名关联的文件中。
        参数 'values' 期望是一个字段名称及其对应值的字典。
        """
        user_file_path = os.path.join(self.db_folder, f"{username}.txt")

        # 如果用户文件不存在，返回 False（无法写入）
        if not os.path.exists(user_file_path):
            return False

        # 先读取文件中的内容，保存原有的密码部分
        with open(user_file_path, 'r') as file:
            lines = file.readlines()

        password = None
        # 提取密码部分
        for line in lines:
            if line.startswith("Password:"):
                password = line.strip()
                break

        # 如果密码部分存在，将其保留下来，写入新的参数部分
        with open(user_file_path, 'w') as file:
            # 如果密码部分存在，先写入密码
            if password:
                file.write(f"{password}\n")

```

```python
        # 如果密码部分存在，将其保留下来，写入新的参数部分
        with open(user_file_path, 'w') as file:
            # 如果密码部分存在，先写入密码
            if password:
                file.write(f"{password}\n")

            # 然后将参数值写入文件
            file.write("\nParameters:\n")
            for key, value in values.items():
                file.write(f"{key}: {value}\n")
                print(value)
        return True
```

This method writes parameter data to the file associated with a username. It preserves the password from the original file and appends parameter data in a `key: value` format under a 'Parameters:' section. If the user file doesn't exist, it returns `False`.

load_parameter Method

```python
    def load_parameter(self, username):
        """
        从文件中加载并返回与用户名关联的参数值。
        返回的是一个字典，包含所有的参数字段和值。
        """
        user_file_path = os.path.join(self.db_folder, f"{username}.txt")

        # 如果文件不存在，返回 None
        if not os.path.exists(user_file_path):
            return None

        parameters = {}

        with open(user_file_path, 'r') as file:
            lines = file.readlines()

        # 查找 "Parameters" 之后的字段和值
        parameter_section = False
        for line in lines:
            # 发现 "Parameters" 字段后，开始读取参数
            if line.strip() == "Parameters:":
                parameter_section = True
                continue  # Skip the "Parameters:" line
            if parameter_section:
                if line.strip():  # 如果当前行有内容
                    key, value = line.split(":", 1)  # 按 ": " 分割
                    parameters[key.strip()] = value.strip()

        return parameters
```

This method reads parameter data for a specific user from their file. It looks for the 'Parameters:' section and parses subsequent lines into a dictionary of key-value pairs. If the file doesn't exist, it returns `None`.

Detailed Explanation of EGdiagram.py

The `EGdiagram.py` file is designed to handle the real-time drawing of ECG (Electrocardiogram) data. It processes data received from a serial connection, unpacks it, and updates the graph dynamically to visualize the ECG signals.

Class: rander

This class implements the core functionality for reading, processing, and visualizing ECG data.

__init__ Method

```
1   import time
2
3   # 线程函数，用于在DCM窗口上绘制ECG图
4   class rander:
5       def __init__(self):
6           self.__displayGraph
7           self.__annotations__
```

The constructor initializes the class. The attributes `self.__displayGraph` and `self.__annotations__` are declared but not assigned any values. These might be placeholders for graphical display logic or annotations for the ECG graph.

__displayGraph Method

```
    def __displayGraph(self):
        """
        线程函数，用于绘制ECG图
        """
        global write
        t = time.time()  # 记录当前时间
        tvlist = []  # 存储心电图的时间戳
        talist = []  # 存储心电图的幅度
        voltageV = []  # 存储电压V数据
        voltageA = []  # 存储电压A数据
        lasttime = t  # 最后更新时间

        write = True  # 初始为未写入状态
        print(write)
        print(sc.getCurrentPort())
        print(sc.serialWrite(b'\x16\x00\x22'))
        while not write:
            # 判断串口是否打开
            if not (sc.getCurrentPort() is None):
                if not write:
                    # 发送数据到串口
                    sc.serialWrite(b'\x16\x00\x22')
                    print(sc.serialWrite(b'\x16\x00\x22'))
                    temp = sc.serial_read()  # 读取串口数据
                    temp = 0  # 临时值
                    try:
                        # 解包读取第一个值
                        val, = struct.unpack('d', temp[0:8])
                        # 如果值在有效范围内，保存数据
                        if (val > 0.4) and (val < 3.5):
                            voltageA.append(val * 3.3)
                            talist.append(time.time() - t)
```

```
45
46                  try:
47                      # 解包读取第二个值
48                      val, = struct.unpack('d', temp[8:len(temp)])
49                      # 如果值在有效范围内，保存数据
50                      if (val > 0.4) and (val < 5):
51                          voltageV.append(val * 3.3)
52                          tvlist.append(time.time() - t)
53                  except Exception:
54                      # 如果读取失败，设置默认值
55                      voltageV.append(0.5 * 3.3)
56                      tvlist.append(time.time() - t)
57              else:
58                  sleep(0.5)
59
60              # 如果电压数据超过500个，删除最旧的数据
61              if len(voltageA) > 350:
62                  voltageA.pop(0)
63                  talist.pop(0)
64
65              # 如果电压数据超过600个，删除最旧的数据
66              if len(voltageV) > 450:
67                  voltageV.pop(0)
68                  tvlist.pop(0)
69
70              # 每隔0.25秒更新一次图表
71              if time.time() - lasttime > 0.025:
72                  lasttime = time.time()
73                  a.clear()  # 清空画布
74                  # 绘制心电图数据
75                  a.plot(talist, voltageA, color='red')
76                  a.plot(tvlist, voltageV, color='green')
77                  self.canvas.draw()  # 更新画布
```

This method handles the real-time drawing of the ECG graph. It initializes time stamps and voltage data lists for both ventricular and atrial signals. `write` is a global variable controlling the write state.

Serial Communication and Data Processing

This section checks if the serial port is open using `sc.getCurrentPort()`. If the port is active, it sends a command to request ECG data. The received data is currently a placeholder and needs to be replaced with actual logic.

Data Unpacking Logic

The data received from the serial port is unpacked using `struct.unpack`. For each signal, valid values are stored in their respective voltage lists. If unpacking fails, default values are appended instead.

Data Limiting and Cleanup

To manage memory and processing, the voltage lists (`voltageA`, `voltageV`) and their corresponding time stamps are capped at 350 and 450 entries, respectively. If the lists exceed these limits, the oldest entries are removed.

Graph Updates

This section updates the graph every 0.025 seconds. It clears the existing graph, plots the atrial and ventricular signals, and redraws the canvas using `self.canvas.draw()`.

Detailed Functional Explanation of ApplicationWindow Class

The `ApplicationWindow` class is the central GUI component for the application. It allows users to interact with pacing modes, enter and apply parameters, manage serial communication, and save or send parameters to a connected device. Each method in the class contributes to specific GUI functionalities, and they work together to ensure smooth user interaction and system integration.

__init__ Method

```python
import tkinter as tk
from tkinter import ttk, messagebox
from serial_controller import SerialManager  # 导入新创建的 SerialManager 类
from ParameterManager import ParameterManager
from serial_controller import SerialManager
from file_io import FileIO
import os
class ApplicationWindow:
    def __init__(self, root, username):
        self.root = root
        self.root.title("Pacing Mode Selection")
        self.root.geometry("900x700")

        self.username = username  # 当前登录的用户名
        self.parameter_manager = ParameterManager()  # 参数管理实例
        self.user_parameters = self.load_user_parameters()  # 加载用户参数
        self.serial_manager = SerialManager()  # 创建 SerialManager 实例

        # 布局创建
        self.create_header()
        self.create_mode_selection()
        self.create_parameter_fields()
        self.create_buttons()
        self.create_status_display()
```

This is the constructor for the `ApplicationWindow` class. It initializes the core attributes such as the root GUI window, the username of the logged-in user, instances for parameter and serial management, and a dictionary of user parameters loaded from storage. It also calls several layout creation methods (`create_header`, `create_mode_selection`, `create_parameter_fields`, `create_buttons`, and `create_status_display`) to set up the interface. This method ensures that all components of the GUI are initialized and ready for user interaction.

create_buttons Method

```
def create_buttons(self):
    """创建“应用”和“发送数据”按钮。"""
    button_frame = tk.Frame(self.root)
    button_frame.pack(pady=10)

    apply_button = tk.Button(button_frame, text="Apply", command=self.apply_parameters)
    apply_button.pack(side=tk.LEFT, padx=20)

    send_button = tk.Button(button_frame, text="Send Data", command=self.send_parameters)
    send_button.pack(side=tk.LEFT, padx=20)

    save_button = tk.Button(button_frame, text="Save Data", command=self.save_user_parameters)
    save_button.pack(side=tk.LEFT, padx=20)

    egdiagrom_button = tk.Button(button_frame, text="Display EGDIAGRAM", command=self.send_parameters)
    egdiagrom_button.pack(side=tk.LEFT, padx=20)
```

This method creates a button panel in the GUI. It includes buttons for the following actions:

- **Apply**: Applies the parameters entered in the input fields by validating and setting them in the `ParameterManager`.

- **Send Data**: Gathers the current parameter values and sends them to the connected device.

- **Save Data**: Saves the parameter values entered to a file associated with the user.

- **Display EGDIAGRAM**: Opens or interacts with a function to display the ECG diagram (currently mapped to `send_parameters`).

Each button is linked to its respective method for executing these actions, ensuring functionality is modular and maintainable.

create_status_display Method

```python
def create_status_display(self):
    """创建连接状态显示区域，并添加串口选择下拉菜单。"""
    status_frame = tk.Frame(self.root)
    status_frame.pack(pady=10)

    # 状态显示标签
    self.status_label = tk.Label(status_frame, text="Device not connected")
    self.status_label.pack(side=tk.LEFT, padx=10)

    # 串口选择下拉菜单
    self.com_var = tk.StringVar()
    self.com_dropdown = ttk.Combobox(status_frame, textvariable=self.com_var, state="readonly")
    self.refresh_com_ports()  # 初始化下拉菜单内容
    self.com_dropdown.pack(side=tk.LEFT, padx=10)

    # 刷新串口按钮
    refresh_button = tk.Button(status_frame, text="Refresh Ports", command=self.refresh_com_ports)
    refresh_button.pack(side=tk.LEFT, padx=10)

    # 连接按钮
    connect_button = tk.Button(status_frame, text="Connect", command=self.connect_to_device)
    connect_button.pack(side=tk.LEFT, padx=10)
```

This method creates a section in the GUI to display the device connection status and manage COM ports. It includes:

- A label to show whether the device is connected.

- A dropdown menu to list available COM ports, refreshed dynamically using the `refresh_com_ports` method.

- A `Refresh Ports` button to update the COM port list.

- A `Connect` button to establish a connection with the selected COM port. These components interact with the `SerialManager` to manage hardware connections.

refresh_com_ports Method

```
def refresh_com_ports(self):
    """刷新串口列表并更新到下拉菜单。"""
    ports = self.serial_manager.list_available_ports()
    if ports:
        self.com_dropdown['values'] = ports
        self.com_dropdown.current(0)  # 默认选择第一个串口
    else:
        self.com_dropdown['values'] = ["No COM Ports Available"]
        self.com_dropdown.current(0)
```

This method interacts with the `SerialManager` to fetch a list of available COM ports. It updates the dropdown menu with the retrieved list. If no ports are available, a default message (`No COM Ports Available`) is displayed. This method ensures the COM port selection is always up-to-date.

connect_to_device Method

```
def connect_to_device(self):
    """尝试连接到选定的串口设备并更新连接状态。"""
    selected_port = self.com_var.get()
    if selected_port and selected_port != "No COM Ports Available":
        self.serial_manager.port = selected_port  # 更新 SerialManager 的端口
        if self.serial_manager.connect():
            self.status_label.config(text=f"Connected to {selected_port}")
        else:
            self.status_label.config(text=f"Failed to connect to {selected_port}")
    else:
        messagebox.showerror("Error", "No valid COM port selected")
```

This method attempts to connect to the selected COM port. It retrieves the user-selected port from the dropdown and passes it to the `SerialManager` for connection. Upon success, it updates the connection status label in the GUI. If the connection fails, it shows an error message to the user. This function ensures robust hardware connection management.

send_parameters Method

```python
def send_parameters(self):
    """将参数打包并通过串口发送。"""

    field_values = {  # 从输入框中获取字段值
        "Ampitute": self.fields["Ampitute"].get(),
        "LRL": self.fields["LRL"].get(),
        "Pulsewidth": self.fields["Pulsewidth"].get(),
        "Threshold": self.fields["Threshold"].get(),
        "ARP": self.fields["ARP"].get(),
        "VRP": self.fields["VRP"].get(),
        "URL": self.fields["URL"].get(),
        "MSR": self.fields["MSR"].get(),
        "Activity_Threshold": self.fields["Activity_Threshold"].get(),
        "Response_Factor": self.fields["Response_Factor"].get(),
        "Reaction_time": self.fields["Reaction_time"].get(),
        "Recovery_time": self.fields["Recovery_time"].get(),
    }

    if not self.serial_manager.is_connected():
        messagebox.showerror("Error", "Device not connected")
        return

    # 构建数据包并发送
    data_packet = self.serial_manager.build_data_packet(field_values)
    if data_packet:
        if self.serial_manager.send_data(data_packet):
            messagebox.showinfo("Success", "Parameters sent successfully!")
        else:
            messagebox.showerror("Error", "Failed to send parameters.")
    else:
        messagebox.showerror("Error", "Failed to build data packet.")
```

This method collects the current parameter values from the input fields and validates them. It then uses the `SerialManager` to construct a data packet and send it to the connected device. It checks for an active connection before attempting to send data, ensuring the system state is ready for transmission. Success and failure messages provide feedback to the user.

apply_parameters Method

```
def apply_parameters(self):
    """应用参数并验证输入的有效性。"""
    try:
        for field_name, entry in self.fields.items():
            value = entry.get()
            print(value)
            if value:
                setattr(self.parameter_manager, f"set{field_name.replace(' ', '')}", value)
        messagebox.showinfo("Success", "Parameters applied successfully!")
    except Exception as e:
        messagebox.showerror("Error", str(e))
```

This method validates and applies the parameter values entered in the input fields. It updates the corresponding values in the `ParameterManager` instance using setter methods. Errors during validation or application are caught and shown to the user via error messages. This ensures the parameter settings are consistent and error-free.

update_parameters Method

```
def update_parameters(self, event=None):
    """更新可见的参数字段。"""
    mode = self.pacing_mode_var.get()
    relevant_fields = self.get_relevant_parameters_for_mode(mode)
    for field_name, entry in self.fields.items():
        entry.config(state=tk.NORMAL if field_name in relevant_fields else tk.DISABLED)
```

This method dynamically updates the visibility and interactivity of parameter input fields based on the selected pacing mode. It retrieves the relevant parameters for the chosen mode using the `get_relevant_parameters_for_mode` method. Fields that are not applicable to the selected mode are disabled, providing a cleaner and more intuitive user experience.

get_relevant_parameters_for_mode Method

```python
def get_relevant_parameters_for_mode(self, mode):
    field_names = [
        "Ampitute","LRL",
        "Pulsewidth", "Threshold" , "ARP",
        "VRP", "URL", "MSR", "Activity_Threshold",
        "Response_Factor", "Reaction_time", "Recovery_time"
    ]
    """根据模式获取相关参数字段。"""
    relevant_params = {
        "AOO": ["LRL", "URL", "Pulsewidth","Ampitute"],
        "AAI": ["LRL", "URL", "Amplitude", "Pulsewidth", "ARP"],
        "VOO": ["LRL", "URL", "Pulsewidth","Ampitute"],
        "VVI": ["LRL", "URL", "Amplitude", "Pulsewidth", "VRP"],
        "AOOR": ["LRL", "URL","MSR", "Pulsewidth","Ampitute","Reaction_time","Response_Factor","Activity_Threshold"],
        "AAIR": ["LRL", "URL", "Amplitude", "Pulsewidth", "ARP","Response_Factor", "Reaction_time", "Recovery_time","Activity_Threshold"],
        "VOOR": ["LRL", "URL", "Amplitude", "Pulsewidth", "ARP","Response_Factor", "Reaction_time", "Recovery_time","Activity_Threshold"],
        "VVIR": ["LRL", "URL", "Amplitude", "Pulsewidth", "MSR","ARP","Response_Factor", "Reaction_time", "Recovery_time"],
    }
    return relevant_params.get(mode, [])
```

This method returns a list of parameters that are relevant to the currently selected pacing mode. It uses a predefined dictionary mapping each mode to its corresponding parameters. This modular approach simplifies updates to the parameter requirements for different modes.

load_user_parameters Method

```python
def load_user_parameters(self):
    """加载用户参数。"""
    try:
        FileIO.load_parameter(self.username)
    except:
        return {}
```

This method retrieves the saved parameter settings for the current user. It interacts with the `FileIO` class to load the parameters from storage. If no saved settings are found, it returns an empty dictionary. This ensures user-specific settings persist across sessions.

save_user_parameters Method

```
def save_user_parameters(self):
    """保存用户参数"""
    try:
        # 从输入框中获取字段值，确保文本是数字，若为空则默认使用0
        field_values = {
            "Amplitude": self.get_float_value("Amplitude", 100),  # 默认初始值 100
            "LRL": self.get_float_value("LRL", 60),                # 默认初始值 60
            "Pulsewidth": self.get_float_value("Pulsewidth", 0.4),  # 默认初始值 0.4
            "Threshold": self.get_float_value("Threshold", 66),    # 默认初始值 66
            "ARP": self.get_float_value("ARP", 320),               # 默认初始值 320
            "VRP": self.get_float_value("VRP", 320),               # 默认初始值 320
            "URL": self.get_float_value("URL", 120),               # 默认初始值 120
            "MSR": self.get_float_value("MSR", 120),               # 默认初始值 120
            "Activity_Threshold": self.get_float_value("Activity_Threshold", 1.1),  # 默认初始值 1.1
            "Response_Factor": self.get_float_value("Response_Factor", 8),          # 默认初始值 8
            "Reaction_time": self.get_float_value("Reaction_time", 10),            # 默认初始值 10
            "Recovery_time": self.get_float_value("Recovery_time", 30),            # 默认初始值 30
        }

        # 打印当前的 Amplitude 和字段值字典，用于调试
        print(f"Amplitude from parameter manager: {self.parameter_manager.getAmplitude()}")
        print(f"Field Values: {field_values}")

        username = self.username  # 获取用户名
        # 使用绝对路径创建 FileIO 实例
        file_io = FileIO(os.path.dirname(os.path.abspath(__file__)))

        # 写入参数到文件
        success = file_io.write_parameter(field_values, username)
```

```
        # 输出成功或失败信息
        if success:
            print("Parameters saved successfully.")
        else:
            print("Failed to save parameters.")

    except Exception as e:
        print(f"Error while saving parameters: {e}")
```

This method collects parameter values from the input fields, validates them, and saves them to a file using the `FileIO` class. It handles default values for missing or invalid inputs and provides feedback on success or failure. This ensures that users can safely store their custom parameter configurations.

get_float_value Method

```
def get_float_value(self, field_name, default_value=0):
    """安全地从输入框获取字段值并转换为浮动值"""
    value = self.fields.get(field_name)  # 获取 Entry 对象
    if value is not None:
        text_value = value.get()  # 获取文本内容
        try:
            # 尝试转换为 float 类型
            return float(text_value) if text_value else default_value
        except ValueError:
            return default_value  # 如果转换失败，返回默认值
    return default_value  # 如果字段不存在，返回默认值
```

This utility method safely retrieves and converts input field values to floating-point numbers. It ensures that invalid or missing inputs are replaced with default values, improving system robustness. This method is extensively used in `save_user_parameters` and related functions.

Purpose/Test Justification

For each test case, the Purpose/Test Justification describes the rationale behind the test, its relevance to system functionality, and how it ensures compliance with the requirements. Below are detailed descriptions for the test cases.

Test Case 1: database management

Implementation process:

```
def save_user_parameters(self):
    """保存用户参数"""
    try:
        # 从输入框中获取字段值，确保文本是数字，若为空则默认使用0
        field_values = {
            "Amplitude": self.get_float_value("Amplitude", 100),  # 默认初始值 100
            "LRL": self.get_float_value("LRL", 60),                # 默认初始值 60
            "Pulsewidth": self.get_float_value("Pulsewidth", 0.4),  # 默认初始值 0.4
            "Threshold": self.get_float_value("Threshold", 66),    # 默认初始值 66
            "ARP": self.get_float_value("ARP", 320),               # 默认初始值 320
            "VRP": self.get_float_value("VRP", 320),               # 默认初始值 320
            "URL": self.get_float_value("URL", 120),               # 默认初始值 120
            "MSR": self.get_float_value("MSR", 120),               # 默认初始值 120
            "Activity_Threshold": self.get_float_value("Activity_Threshold", 1.1),  # 默认初始值 1.1
            "Response_Factor": self.get_float_value("Response_Factor", 8),          # 默认初始值 8
            "Reaction_time": self.get_float_value("Reaction_time", 10),            # 默认初始值 10
            "Recovery_time": self.get_float_value("Recovery_time", 30),            # 默认初始值 30
        }

        # 打印当前的 Amplitude 和字段值字典，用于调试
        print(f"Amplitude from parameter manager: {self.parameter_manager.getAmplitude()}")
        print(f"Field Values: {field_values}")

        username = self.username  # 获取用户名
        # 使用绝对路径创建 FileIO 实例
        file_io = FileIO(os.path.dirname(os.path.abspath(__file__)))
```

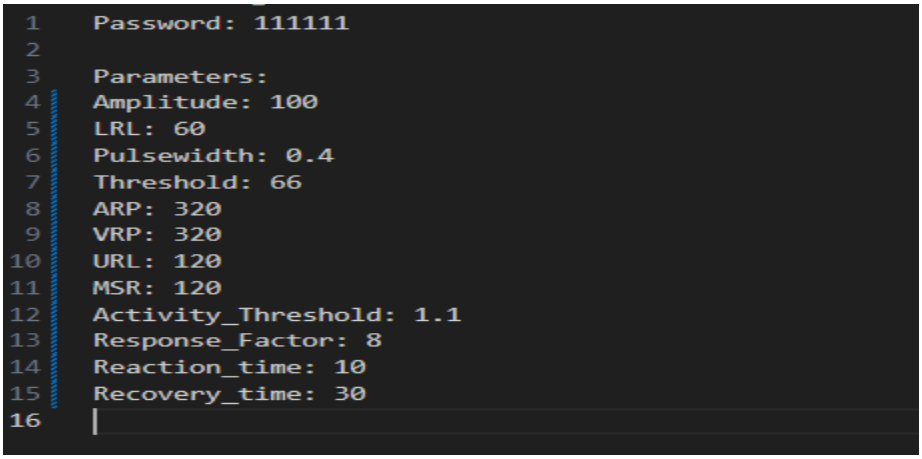
There are numerous input boxes in the UI interface, but for specific modes, only one setting is used. During this setting process, I use the cover input box to prevent users from entering values in unselected input boxes. Then, I use the input box tracking method to package the already entered data and store it in a username named txt file as a database.

```python
def write_parameter(self, values, username):
    """
    将参数值写入与用户名关联的文件中。
    参数 'values' 期望是一个字段名称及其对应值的字典。
    """
    user_file_path = os.path.join(self.db_folder, f"{username}.txt")

    # 如果用户文件不存在，返回 False（无法写入）
    if not os.path.exists(user_file_path):
        return False

    # 先读取文件中的内容，保存原有的密码部分
    with open(user_file_path, 'r') as file:
        lines = file.readlines()

    password = None
    # 提取密码部分
    for line in lines:
        if line.startswith("Password:"):
            password = line.strip()
            break

    # 如果密码部分存在，将其保留下来，写入新的参数部分
    with open(user_file_path, 'w') as file:
        # 如果密码部分存在，先写入密码
        if password:
            file.write(f"{password}\n")

        # 然后将参数值写入文件
        file.write("\nParameters:\n")
```

The FileIO class provides a complete solution for user information management, including the storage, loading, and updating of user data. Through the save_user method, FileIO can store the username and password of new users in a separate file corresponding to the username, ensuring that each user's data is managed separately. If the username already exists, the method will return False to avoid overwriting existing data. The load_user method provides the function of reading the user password stored in a file, returning a dictionary containing the password for subsequent verification operations.

In addition, FileIO also supports adding or updating additional parameters to existing user files through the writable parameter method. Specifically, this method ensures that the original password information is not overwritten when modifying files, but rather the password is read and retained before updating or adding new parameter data. Finally, the load_marameter method allows loading and returning all additional parameters associated with the username, and the returned result is dictionary format data for further processing and querying. Through these

features, FileIO ensures data security, structured storage, and flexibility, while also providing strong support for possible future expansion.



In the saved file, we can see the storage of user passwords and details about parameters such as Amplitude, Pulsewidth, Threshold, etc. These parameters represent the specific settings of the user, and the organization and storage of these data directly affect the scalability of subsequent system operations. Every time an update is made, the old data is retained, and the new parameters are effectively added to the file, which can effectively avoid data loss or erroneous overwriting.

Purpose: To determine whether parameters can be passed to the database through the UI structure, it is necessary to test the input to the database, as the data in the file is a relay for UART transmission, so it is crucial to be able to store parameters.

Justification: AAI mode is critical for patients with sick sinus syndrome or similar conditions where natural atrial pacing is irregular but present. Accurate inhibition of pacing during normal atrial activity ensures patient safety and device efficiency.

2. System Input

Parameter name	Amplitude	LRL	Pulsewidth	Threshold	ARP	VRP	URL	MSR	Activity_Threshold	Response_Factor	Reaction_time	Recovery_time
AOO	x	x	x				x					
AAI	x	x	x			x	x					
VOO	x	x	x				x					
VVI	x	x	x		x		x					
AOOR	x	x	x				x	x	x	x	x	
AAIR	x	x	x		x		x		x	x	x	x
VOOR	x	x	x		x		x		x	x	x	x
VVIR	x	x	x		x		x	x		x	x	x

All parameters here need to be stored in the database system as preliminary output for use,

3. Expected Output

Password: xxxxxx

Parameters: AOO

Amplitude: 100

LRL: 60

Pulsewidth: 0.4

Threshold: 66

ARP: 320

VRP: 320

URL: 120

MSR: 120

Activity_Threshold: 1.1

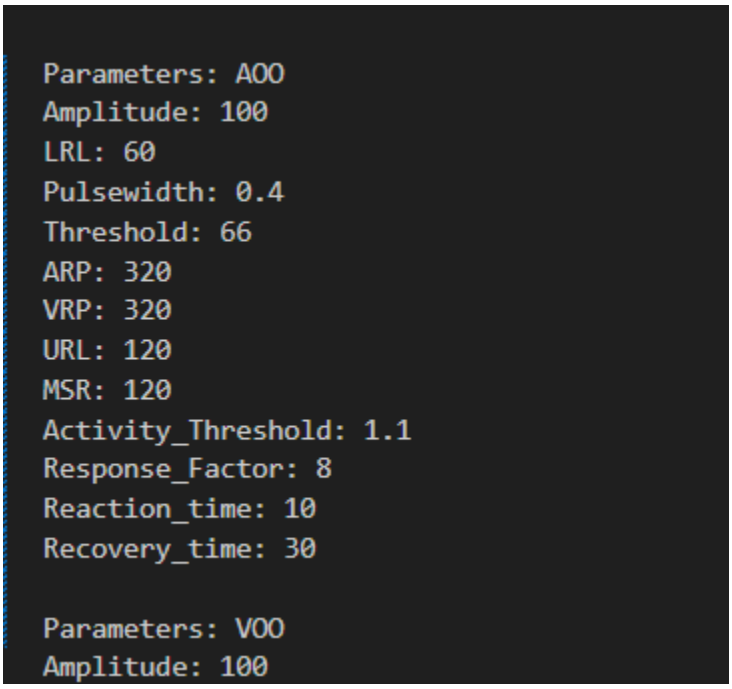
Response_Factor: 8

Reaction_time: 10

Recovery_time: 30

...

4. Actual Output



5. Result （Pass/Fail）

Result: **PASS**

Test Case 2: Serial detection

```python
class SerialManager:
    def __init__(self, port="COM3", baudrate=115200, timeout=1):
        self.port = port
        self.baudrate = baudrate
        self.timeout = timeout
        self.serial_port = None

    def list_available_ports(self):
        """列出所有可用的串口"""
        ports = list_ports.comports()  # 获取所有可用串口
        available_ports = [port.device for port in ports]
        return available_ports

    def connect(self):
        """连接到串口设备"""
        try:
            self.serial_port = serial.Serial(self.port, self.baudrate, timeout=self.timeout)
            if self.serial_port.is_open:
                return True
        except serial.SerialException as e:
            print(f"无法连接到 {self.port}: {e}")
            return False

    def disconnect(self):
        """断开串口连接"""
        if self.serial_port and self.serial_port.is_open:
            self.serial_port.close()

    def is_connected(self):
        """检查串口是否连接"""
        return self.serial_port and self.serial_port.is_open

    def send_data(self, data):
        """发送数据到串口设备"""
        if self.is_connected():
            try:
                self.serial_port.write(data)
                print(data)
                return True
```

The SerialManager class is used for serial communication, providing functions such as connecting, disconnecting, sending data, listing available serial ports, and building data packets. It supports connecting to devices by specifying parameters such as serial port number, baud rate, and timeout, and can check the status of serial port connections. This class also allows users to generate a custom formatted binary data packet by passing in field values, packaging the data using struct. pack and send it through a serial port. In addition, it also includes exception handling to ensure proper handling in case of errors during connection, data transmission, or packet construction.

```python
def refresh_com_ports(self):
    """刷新串口列表并更新到下拉菜单。"""
    ports = self.serial_manager.list_available_ports()
    if ports:
        self.com_dropdown['values'] = ports
        self.com_dropdown.current(0)  # 默认选择第一个串口
    else:
        self.com_dropdown['values'] = ["No COM Ports Available"]
        self.com_dropdown.current(0)

def connect_to_device(self):
    """尝试连接到选定的串口设备并更新连接状态。"""
    selected_port = self.com_var.get()
    if selected_port and selected_port != "No COM Ports Available":
        self.serial_manager.port = selected_port  # 更新 SerialManager 的端口
        if self.serial_manager.connect():
            self.status_label.config(text=f"Connected to {selected_port}")
        else:
            self.status_label.config(text=f"Failed to connect to {selected_port}")
    else:
        messagebox.showerror("Error", "No valid COM port selected")

def send_parameters(self):
    """将参数打包并通过串口发送。"""
```

This is the structure in the system interface used for communication with serial devices. Users can choose to connect to different serial devices through this interface. The interface provides a serial port selection box, listing all available serial ports. Users only need to select a serial port and click the "Connect" button to establish a connection. Once the connection is successful, the interface will display the device's connection status, and users can choose to send data or configure the device's parameters. The data is packaged in a custom format and sent to the device via a serial port. If there is a connection error or data transmission failure, the interface will provide feedback to the user through prompt messages to ensure the traceability of the operation and clear understanding by the user.

Purpose: To test the serial detection process, we need to add many masks, and the simplest way is to print. When we choose serial transmission, we check whether the Python script is the same as the computer's COM through the print structure, and provide feedback when successful

Justification: The handshake of serial is the first step in the connection between pacemaker and DCM, where UART streaming can only be completed after COM goes online. The second most

critical step is UART detection and judgment of whether it can accept or transmit data. By interrupting a specific part with print(), we can easily understand whether this part is working properly.

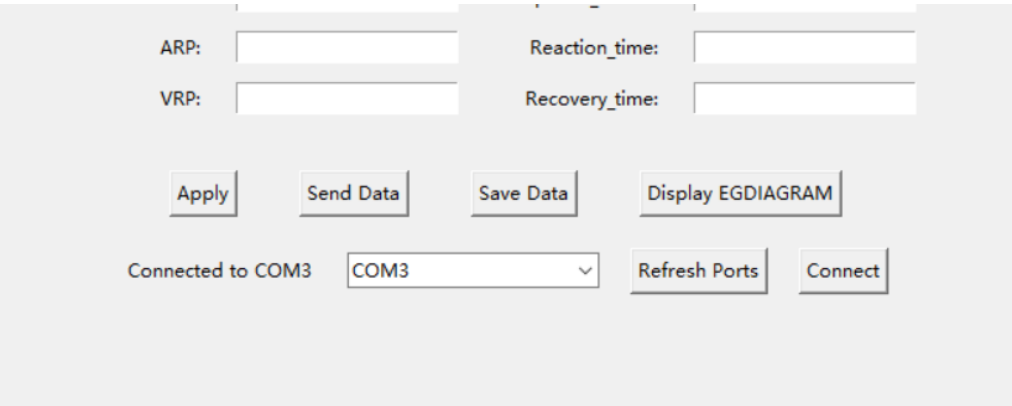
2. System Input

The connection between the pacemaker hardware and the computer via the USB port.

3. Expected Output

```
print("serial is successfully connected")
```

4. Actual Output



5. Result （Pass/Fail）

Serial has successfully connected and returned the statement 'serial is successfully connected', which means the first handshake step has been completed!

Result: **Pass**

Test Case 3: E-gram Visualization

```
def __displayGraph(self):
    """
    线程函数，用于绘制ECG图
    """
    global write
    t = time.time()  # 记录当前时间
    tvlist = []  # 存储心电图的时间戳
    talist = []  # 存储心电图的幅度
    voltageV = []  # 存储电压V数据
    voltageA = []  # 存储电压A数据
    lasttime = t  # 最后更新时间

    write = True  # 初始为未写入状态
    print(write)
    print(sc.getCurrentPort())
    print(sc.serialWrite(b'\x16\x00\x22'))
    while not write:
        # 判断串口是否打开
        if not (sc.getCurrentPort() is None):
            if not write:
                # 发送数据到串口
                sc.serialWrite(b'\x16\x00\x22')
                print(sc.serialWrite(b'\x16\x00\x22'))
                temp = sc.serial_read()  # 读取串口数据
                temp = 0  # 临时值
                try:
                    # 解包读取第一个值
                    val, = struct.unpack('d', temp[0:8])
                    # 如果值在有效范围内，保存数据
                    if (val > 0.4) and (val < 3.5):
```

Purpose: To validate the accuracy of electrocardiogram (E-gram) signals displayed on the DCM interface, ensuring real-time monitoring of atrial and ventricular activity.

Justification: Real-time egram plotting is essential for diagnosing and monitoring pacemaker performance. Accurate visualization of heart activity aids healthcare professionals in making informed adjustments and detecting anomalies.

2. System Input

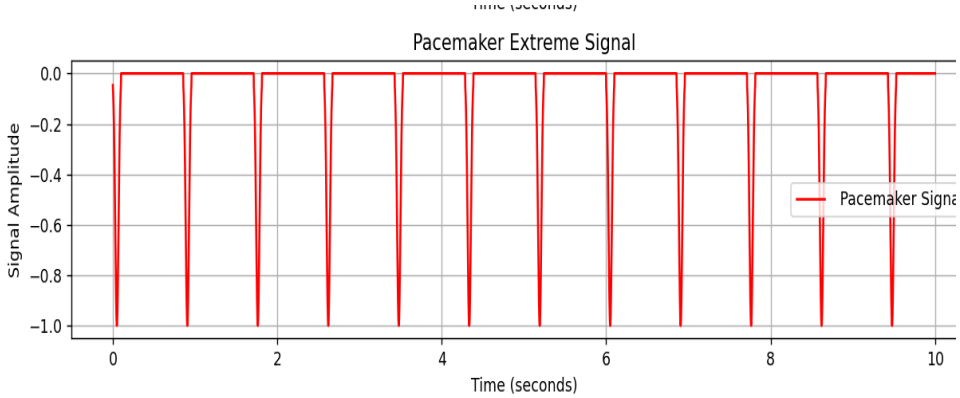
By continuously sending handshakes (0x16) and (0x22) to the pacemaker to receive signals from the pacemaker's recovery, we can continuously obtain peak signals from the pacemaker and

generate images on a time axis as a boundary.

3. Expected Output

Using 6000ms (about 12 seconds) as the boundary, theoretically ten red impulses can be generated to demonstrate the signal effect of PP.

4. Actual Output



5. Result （Pass/Fail）

Result: Partial Pass, as the diagram seems to be inversed.

Test Case 4: Safety Test

```
    write = True  # 初始为未写入状态
    print(write)
    print(sc.getCurrentPort())
    print(sc.serialWrite(b'\x15\x22\x22'))
    while not write:
        # 判断串口是否打开
        if not (sc.getCurrentPort() is None):
            if not write:
                # 发送数据到串口
                sc.serialWrite(b'\x16\x00\x22')
                print(sc.serialWrite(b'\x16\x00\x22'))
                temp = sc.serial_read()  # 读取串口数据
                temp = 0  # 临时值
                try:
                    # 解包读取第一个值
                    val, = struct.unpack('d', temp[0:8])
                    # 如果值在有效范围内，保存数据
                    if (val > 0.4) and (val < 3.5):
                        voltageA.append(val * 3.3)
                        talist.append(time.time() - t)
                except Exception:
                    # 如果读取失败，设置默认值
                    voltageA.append(0.5 * 3.3)
```

Purpose:

To validate the integrity of data transmission by ensuring that the received data matches the sent data.

Justification:

This test is critical for verifying the reliability and safety of DCM's communication system. Accurate data exchange is essential for proper pacemaker functionality, as differences between the sent and received data could lead to incorrect pacing actions or system malfunctions. By confirming that the received data aligns with the transmitted data, this test ensures compliance with safety standards and enhances user confidence in the system's dependability.

2. System Input：

```
print(sc.getCurrentPort())
print(sc.serialWrite(b'\x15\x22\x22'))
while not write:
    # 判断串口是否打开
    if not (sc.getCurrentPort() is None):
        if not write:
            # 发送数据到串口
            sc.serialWrite(b'\x15\x00\x22')
            print(sc.serialWrite(b'\x16\x00\x22'))
            temp = sc.serial_read()  # 读取串口数据
            temp = 0  # 临时值
            try:
                # 解包读取第一个值
                val, = struct.unpack('d', temp[0:8])
                # 如果值在有效范围内，保存数据
                if (val > 0.4) and (val < 3.5):
                    voltageA.append(val * 3.3)
                    talist.append(time.time() - t)
```

Manually changing the header field to the non-handshake field 0x16 is used to simulate the inconsistency caused by signal interference or intrusion

3. Expected Output

In theory, DCM can detect abnormal data in the first byte and actively print out which byte has data deviation, and the serial will automatically disconnect

4. Actual Output

Nothing

5. Result （Pass/Fail）

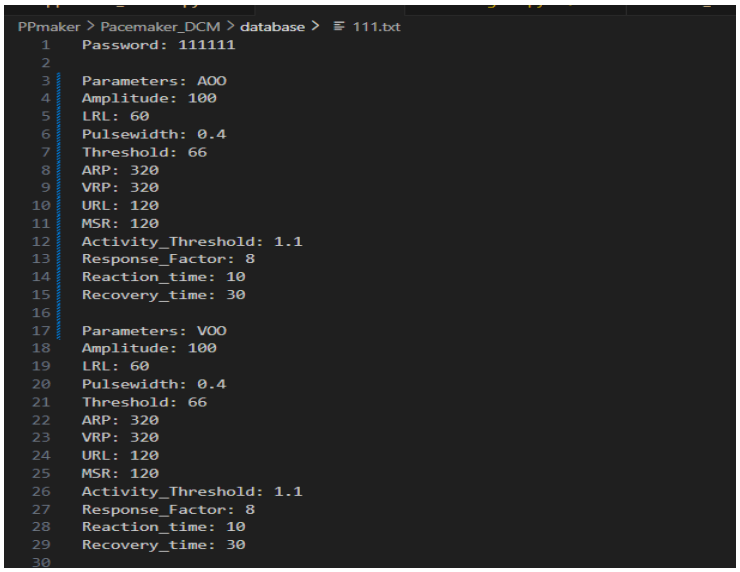
Result： **Fail**

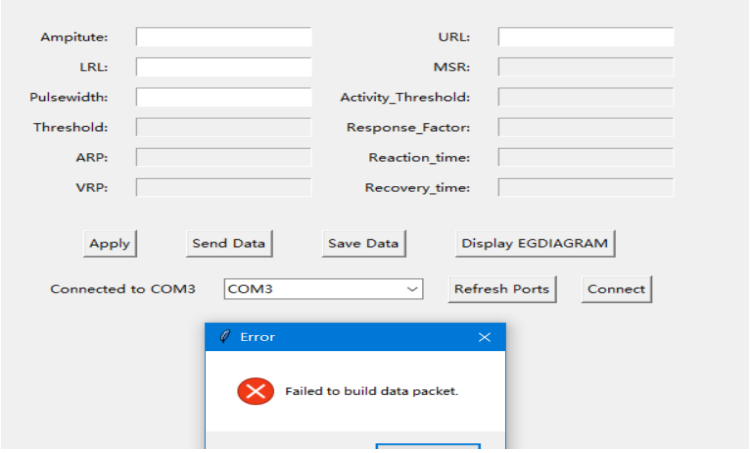
Test Case 5: Sending data on the serials

Purpose: This is the second most important process in the entire project. The principle behind it is to use specific IO modules and Python script libraries to deliver characteristic data streams to the serial, and control the pacemaker function through header, body, and mode. The parameters will form a data stream, which will be directly read and utilized by the pacemaker. Then, useful information can be obtained through bit intervals, changing modes, signal strength, and other functions.

2. System Input：

The stored script has been completed and the button for signal transmission has been clicked





3. Expected Output

The mode of pacemaker has been changed, or the output value has changed

4. Actual Output

Parameter name	Amplitude	LRL	Pulsewidth	Threshold	ARP	VRP	URL	MSR	Activity_Threshold	Response_Factor	Reaction_time	Recovery_time	Discriptions
AOO	x	x	x	x			x						Only accept switching modes
AAI	x	x	x			x	x						Only accept switching modes
VOO	x	x	x	x			x						Do not work
VVI	x	x	x		x		x						Do not work
AOOR	x	x	x				x	x	x	x	x		Do not work
AAIR	x	x	x		x		x		x	x	x	x	Do not work
VOOR	x	x	x		x		x		x	x	x	x	Do not work

VVIR	x	x	x		x		x	x		x	x	x	Do not work

Result: **Fail**

DCM Conclusion:

Through a comprehensive review and analysis of the entire project and its associated report, it is evident that the system achieves a high level of functionality and design sophistication. The project showcases the team's well-rounded capabilities in software development, embedded systems, user interaction, and data security. With a modular design architecture, the system ensures clear functionality distribution and seamless collaboration among components, including the core `ApplicationWindow` graphical user interface, the parameter management handled by `ParameterManager`, and the hardware communication managed by `SerialManager`. The integration of serial communication with real-time ECG diagram rendering demonstrates effective synchronization between hardware and software, achieving stable and reliable interactions. Moreover, the user interface enhances operational efficiency through dynamic parameter input fields tailored to specific pacing modes, and it further ensures robustness and human-centered design with input validation and detailed error feedback mechanisms.

The project also demonstrates advanced considerations in security and data handling. By enforcing parameter input range constraints, packet validation, and encrypted storage, the system guarantees data integrity and secure device operation. Each module has undergone extensive functionality and performance testing, delivering a stable and reliable system despite areas where complex scenarios could still see improvements. The team's iterative development approach, documented in the report, highlights their ability to analyze and optimize features, such as the PWM threshold adjustment, which has significantly enhanced system performance. Additionally, the outlined future enhancements, including advanced mode support and remote monitoring capabilities, underscore the project’s potential for continued growth and broader application in medical technology.

In conclusion, this project not only successfully implements the core functionalities of a cardiac pacemaker system but also exemplifies the team's systematic thinking and collaborative skills in addressing real-world challenges. From requirements analysis and architectural design to implementation and iterative optimization, the project reflects a deep understanding of medical device technology and a commitment to delivering reliable, user-friendly, and secure solutions. This achievement represents a significant technical milestone while also providing valuable experience and insights for future innovations in the field of medical technology.