Deliverable 1: Documentation

MECHTRON 3K04

L04 - Group #7 - Heartware

Oct. 27, 2025

1. Group Members

Member Name	MACIDs	Student Numbers
Zainab Iqbal	iqbalz7	400514865
Maryam Khatib	khatim3	400515768
Min Yi Liu	liu1957	400502725
Alexander Sun	suna44	400510053
Cynthia Sun	sunc44	400238765
Karolina Teresinska	teresink	400535227

2. Part 1

2.1. Introduction

Heartware is a pacemaker – a safety critical implant – which regulates a user's heart rate by sending electrical pulses to the heart's nodes. Such devices are used to treat conditions such as arrhythmia and bradycardia, where a patient's heart is unable to sufficiently regulate its own rate.

To create the Heartware pacemaker, an FRDM-K64F microcontroller unit (MCU) was selected for signal generation and processing. This component interfaces with a shield that amplifies, filters, and rectifies signals. These hardware choices are advantageous, as the MCU contains an accelerometer to allow for rate-adaptive pacing to be implemented in future iterations, as well as a magnetometer which allows for switching of pacing modes. To program the hardware, Simulink and Stateflow were used to design pacing modes and control interaction between modes. Using these model-based design applications allows for easy design, simulation, and testing, while also reflecting the tools real-world safety-critical systems generally use for similar implementations. Stateflow was used to create state transition diagrams which contained internal logic to dictate the pacing pattern of the MCU, while Simulink created a bridge between inputs and outputs to the MCU.

In addition to the embedded system of Heartware, a device control monitor (DCM) was programmed as a desktop graphical user interface (GUI). The DCM will allow registered users to control pacing mode, change parameters such as amplitude, rate, pulse width, and eventually display relevant data. Though it does not currently communicate with the MCU, the DCM implements limits on parameter setting. The DCM ultimately provides the end-user with the capacity to make use of the data detected and transmitted by the MCU. Using various Python libraries, graphics, menus, prompts, and other functionalities could be implemented. JSON files were also critical in the development of the DCM as they provided the means to access the DCM's exports in desktop environments.

2.1.1. AOO/VOO

The AOO and VOO modules are Simulink and Stateflow implementations that provide pacing to the atrial or ventricular chambers of the heart respectively. The 'A' denotes atrial pacing (or 'V' denoting ventricular pacing), while each 'O' represents that chambers are neither being sensed nor is a subsequent response being produced. Through manipulating switches via integrated pins, the heart can be paced through the simultaneous discharging of one capacitor and the charging of another.

2.1.2. AAI/VVI

The AAI and VVI modules are Simulink and Stateflow implementations that provide both pacing and sensing for the atrial or ventricular chambers of the heart respectively. In these modes, the first letter 'A' or 'V' denotes atrial or ventricular pacing respectively, the second letter indicates the sensed chamber and the 'I' indicates that the pacemaker inhibits pacing if an intrinsic beat is sensed during the programmed interval. These modules extend the AOO/VOO modules by using hardware feedback to determine

whether to deliver or inhibit a pulse. Pacing occurs through the charging and discharging of capacitors when no intrinsic beat is sensed.

2.1.3. DCM:

The DCM is a desktop GUI that allows for log in and registration with a maximum of 10 login credentials). The user can then select a pacing mode (AOO/VOO/AAI/VVI), configure pacing parameters within safe ranges, and save/retrieve settings per login user. In Deliverable 1, the implemented components include DCM's software structure, validation, and persistence of saved data. The hardware communication and Egram visualization will be integrated in later deliverables

2.2. Requirements

2.2.1. AOO Mode

The pacemaker in AOO mode must deliver fixed-rate pacing of a set magnitude to the atrium. Additionally, it must ensure pacing does not continue when the magnet is out of place, instead switching to AAI mode, or when previous pacing fails to be consistent. When the pacemaker is in AOO mode, it must not sense or respond to sensing of the atrium, and must deliver all of its pacing independently and asynchronously.

GIVEN variable m_magnet
AND m_magnet = INPLACE
WHEN deciding p_aooInterval and p_aPaceWidth
THEN do not change p_aooInterval and p_aPaceWidth **AND** enter AOO mode

GIVEN device is in AOO mode
AND Periodic(magEvent, p_aooInterval) = true
WHEN deciding c_vp value
THEN c_vp = p_aPaceAmp

GIVEN device is in AOO mode
AND Periodic(magEvent, p_aooInterval) = true
AND time elapsed ≤ p_aPaceWidth
WHEN detecting atrial pacing state
THEN In_aPace = true

GIVEN device is in AOO mode
AND Periodic(magEvent, p_aooInterval) = true
AND time elapsed > p_aPaceWidth
WHEN detecting atrial pacing state
THEN In_aPace = false

GIVEN device is in AOO mode
AND Prior(In_aPace) = true
BUT In_aPace != true
WHEN deciding c_vp value
THEN c_vp = 0

GIVEN device is in AOO mode
AND Periodic(magEvent, p_aooInterval) != true
OR Prior(In_aPace) != true
OR In_aPace != true
WHEN deciding c_vp value
THEN do not change c_vp

2.2.2. VOO Mode

The pacemaker in VOO mode must deliver a specific, fixed-rate pacing of a set magnitude to the ventricle. Additionally, it must ensure that its pacing does not continue when the magnet is out of place, instead switching to VVI mode, or if previous pacing fails to be consistent. When the pacemaker is in VOO mode, it must remain unaffected by sensing of the ventricle, and must deliver all of its pacing independently and asynchronously.

GIVEN variable m_magnet
AND m_magnet = INPLACE
WHEN deciding p_vooInterval and p_vPaceWidth
THEN do not change p_vooInterval and p_vPaceWidth **AND** enter VOO mode

GIVEN device is in VOO mode
AND Periodic(magEvent, p_vooInterval) = true
WHEN deciding c_vp value
THEN c_vp =p_vPaceAmp

GIVEN device is in VOO mode
AND Periodic(magEvent, p_vooInterval) = true
AND time elapsed ≤ p_vPaceWidth
WHEN detecting ventricle pacing state
THEN In_vPace = true

GIVEN device is in VOO mode
AND Periodic(magEvent, p_vooInterval) = true
AND time elapsed > p_vPaceWidth
WHEN detecting ventricle pacing state
THEN In_vPace = false

GIVEN device is in VOO mode

AND Prior(In_vPace) = true
BUT In_vPace != true
WHEN deciding c_vp value
THEN c_vp = 0

GIVEN device is in VOO mode
AND Periodic(magEvent, p_vooInterval) != true
OR Prior(In_vPace) != true
OR In_vPace != true
WHEN deciding c_vp value
THEN do not change c_vp

2.2.3. AAI Mode

In AAI mode, the pacemaker only delivers pacing when the atrium does not produce a natural beat within the required interval. To achieve this, it constantly monitors atrial activity and delivers a pacing pulse of the set amplitude and pulse width only when no natural atrial heartbeat is detected. If atrial activity occurs before the interval is completed, the pacemaker inhibits pacing for that cycle. After every sensed or paced event, a refractory period (ARP) occurs during which sensed signals are ignored and pacing is prevented. The pacemaker remains in AAI mode as long as the magnet is not in place. However, when a magnet is detected (m_magnet = INPLACE), the device switches to AOO mode, overriding sensing and pacing. Thus, AAI mode must deliver demand-based pacing that synchronizes with the patient's intrinsic atrial rhythm while preventing unnecessary or premature pacing stimuli.

GIVEN variables m_magnet and In_Pacing_atrium
AND m_magnet != INPLACE
AND In_Pacing_atrium = true
WHEN deciding p_lowrateInterval and p_aPaceWidth
THEN do not change p_lowrateInterval and p_aPaceWidth AND enter AAI mode

When pacing is needed:
GIVEN device is in AAI mode
AND Periodic(aEvent, p_lowrateInterval) = true
WHEN deciding c_ap behavior
THEN c_ap = p_aPaceAmp

Inhibition when atrial event is sensed:
GIVEN device is in AAI mode
AND Prior(In_Pacing_atrium) = true
BUT m_as = true
WHEN deciding c_ap behavior
THEN c_ap = 0

No change:
GIVEN device is in AAI mode

AND Periodic(aEvent, p_lowrateInterval) != true
OR Prior(In_Pacing_atrium) != true
OR In_Pacing_atrium != true
WHEN deciding c_ap behavior
THEN do not change c_ap

Atrial Refractory Period:
GIVEN device is in AAI mode
AND In_ARP = true
WHEN deciding whether to respond to m_as
THEN ignore m_as until In_ARP = false

If magnet is placed:
GIVEN device is in AAI mode
AND m_magnet = INPLACE
WHEN deciding mode
THEN switch to AOO mode

2.2.4. VVI Mode

In VVI mode, the pacemaker only delivers pacing when the ventricle does not produce a natural beat within the required interval. To achieve this, it constantly monitors ventricular activity and delivers a pacing pulse of the set amplitude and pulse width only when no natural ventricular heartbeat is detected. If ventricular activity occurs before the interval is completed, the pacemaker inhibits pacing for that cycle. After every sensed or paced event, a refractory period (VRP) occurs during which sensed signals are ignored and pacing is prevented. The pacemaker remains in VVI mode as long as the magnet is not in place. However, when a magnet is detected (m_magnet = INPLACE), the device switches to VOO mode, overriding sensing and pacing. Thus, VVI mode must deliver demand-based pacing that synchronizes with the patient's intrinsic rhythm while preventing unnecessary or premature pacing stimuli.

GIVEN variables m_magnet and In_Pacing_ventricle
AND m_magnet != INPLACE
AND In_Pacing_ventricle = true
WHEN deciding duration interval (p_lowrateInterval) and pulse width (p_vPaceWidth)
THEN do not change p_lowrateInterval and p_vPaceWidth AND enter VVI mode

When pacing is needed:
GIVEN device is in VVI mode
AND Periodic(vEvent, p_lowrateInterval) = true
WHEN deciding c_vp behaviour
THEN c_vp = p_vPaceAmp

When the pacemaker is inhibited because it detects a heartbeat soon enough:
GIVEN device is in VVI mode
AND Prior(In_Pacing_ventricle) = true

BUT m_vs = true
WHEN deciding c_vp behavior
THEN c_vp = 0

Refractory Period of the heart:
GIVEN device is in VVI mode
AND VRP_active = true
WHEN deciding whether to respond to m_vs
THEN ignore m_vs until VRP_active = false

Then when the magnet is in place it will switch to VOO
GIVEN device is in VVI mode
AND m_magnet = INPLACE
WHEN deciding mode
THEN switch to VOO mode

2.2.5. DCM

The Device Controller Monitor (DCM) serves as an interactive platform that enables a user to securely view, configure and store programmable parameters associated with a pacemaker operation. Generally, the DCM must provide an interactive and user-friendly interface through which users can select one of the four allowed modes for Deliverable 1, then modify all pacing parameters.

At the highest level, the DCM must display the LoginPage upon launch. This page allows for user registration, authentication, and login, while enforcing the maximum capacity of 10 user accounts. The usernames are treated in a case-insensitive manner and the password field is censored. During registration, the DCM must verify that the username and password fields are not empty, and if capacity permits, it must hash the password before storage. Any attempt to exceed the user capacity, or any attempt to register or login with a missing or mismatching field must be met with a pop-up error message that clearly indicates the cause of the rejection. Successful registration will prompt the user to login.

Upon successful login and authentication, the system must transition to DashboardPage, which displays the current communication, device, and telemetry status of the pacemaker interface. The user must be able to select from four selectable mode options - AOO, VOO, AAI, and VVI. Selecting any mode must open its corresponding ModeEditorPage, where the user can adjust parameters specific to that mode. The programmable parameters supported include Lower Rate Limit (LRL), Upper Rate Limit (URL), Atrial and Ventricular Amplitude, Pulse Width, Refractory Period, Hysteresis and Rate Smoothing. The DCM must ensure that all entered values remain within the prescribed and software validated limits. Additionally, the values that can be edited (can be switched from “Off” to “On”) must depend on the chosen mode. These limits are hidden within the code as global variables and constants that cannot be changed by outside influences. Through this page, the DCM must allow users to save all validated parameters to a local JSON file. These parameters are uniquely associated with the user. Users must also be able to revert back to their last saved parameters after changes.

The DCM must also allow the user to view a Summary (tab) with their chosen parameters, as well as the Egram tab, that is currently set with simulated values but will be updated for Deliverable 2.

The DCM must allow the users to export the parameters as a JSON file and generate printable reports such as Bradycardia and Temporary parameter summaries. This can be done through a File menu. Additionally, the DCM must provide toggles to be able to switch between “Connected” and “Not Connected,” mark a device as “Changed,” set a new device ID, and toggle telemetry between “OK,” “Lost: Out of Range,” and “Lost: Noise.” This can be done through a Simulator menu. There must also be a Utilities tab that has an About page displaying data such as the model number, software revision, serial number and institution, as well as a Set Clock Dialog that allows users to change the displayed time.

As mentioned prior, the DCM must validate all numeric inputs before saving or exporting. Each widget must reject values that are out of range and apply rounding rules to parameters where applicable, The interface must provide feedback messages upon successful saves or exports, and error messages following failures. Following these requirements ensures the DCM can operate as a dependable, medically compliant interface for programming and configuring pacemakers.

2.3. Design Decisions

For the purposes of this documentation, any use of the letter ‘x’ or ‘X’ in variable names represents either ‘a’ or ‘v’ (‘A’ or ‘V’) for atrium or ventricle, respectively. Since the functionality and implementation for the XOO modes are essentially the same, this decision was taken for the sake of clarity. The same is applicable to the XXI modes.

The board used for the pacemaker is an FRDM-K64F. The circuitry on the board helps supply voltage to the heart such that it can take proper actions.

Table 1. The variables used for the implementation of XOO and XXI with their corresponding pins on the board, function, and value range.

Variable Name	**Pin**	**Function**	**Value Range**
PACE_CHARGE_CTRL	D2	Controls the charge of capacitor C22. In its HIGH state, PWM charges C22, and in its LOW state, PWM disconnects from the circuit to discharge.	HIGH/LOW
PACE_GND_CTRL	D10	Controls current flow from the ring to the tip of either the atrium or ventricle. Works with x_PACE_CTRL pins to control charge flow to C21.	HIGH/LOW

ATR_PACE_CTRL	D8	Involved in the discharging of capacitor C22, with HIGH signifying current flow through the switch and LOW signifying no current flow.	HIGH/LOW
VENT_PACE_CTRL	D9		
ATR_GND_CTRL	D11	Connect the circuit to the ground to discharge the blocking capacitor C1 in either the atrium or ventricle.	HIGH/LOW
VENT_GND_CTRL	D12		
Z_ATR_CTRL	D4	Allows the impedance circuit to be connected to the ring electrode of the atrium or ventricle. Analyzes impedance of the electrode as well as assessing the electrical connection between the electrode and the atrium or ventricle.	HIGH/LOW
Z_VENT_CTRL	D7		HIGH/LOW
PACING_REF_PWM	D5	Controls voltage supplied to the capacitor by generating a PWM signal with a duty cycle proportional to the ratio of the desired voltage to the maximum voltage. By increasing the duty cycle, passed into the pin as a value from 1-100, the average voltage delivered to the capacitor increases from 1% to 100% of the maximum output voltage, 5V. The default frequency for the PWM signal is 2000 Hz.	1-100
XXI ONLY PINS			
FRONTEND_CTRL	D13	Activates sensing circuitry. In its HIGH state, the sensing circuitry outputs a heart signal and in its LOW state the sensing circuitry is disconnected (does not output anything).	HIGH/LOW
ATR_CMP_DETECT	D0	Used in the sensing circuitry. In its HIGH state, it outputs ON when the signal voltage > threshold voltage and otherwise (LOW state) it outputs OFF.	HIGH/LOW
VENT_CMP_DETECT	D1		
ATR_CMP_REF_PWM	D6	Establishes a threshold relative to which atrial/ventricular action potential should be sensed.	52
VENT_CMP_REF_PWM	D3		

2.3.1 AOO and VOO Modes

2.3.1.1. System Architecture

The AOO and VOO implementations can be broken down into three main substates: pacing either the atrium or ventricle, discharging the blocking capacitor, and charging the primary capacitor.

Pacing the heart chambers occurs when PACE_GND_CTRL and x_PACE_CTRL are set to HIGH while PACE_CHARGE_CTRL, x_GND_CTRL, and PACING_REF_PWM are set to LOW. The pins are set to their specific values to allow the current to flow through the heart while discharging the primary capacitor from the previous state. This ensures there is no change in capacitance yet, while providing the voltage pacing for the heart.

Discharging the blocking capacitor C21 occurs after the duration of p_vPaceWidth in milliseconds from pacing the heart chamber. This is when x_GND_CTRL switches to HIGH, x_PACE_CTRL switches to LOW, and other pins maintain their original values. Since the current from pacing the heart flows into the blocking capacitor, all the changes in pin value in this state are to allow this current to discharge.

The discharging of blocking capacitor C21 then transitions to the charging of the primary capacitor C22 without any conditions, as they are electrically independent and can occur simultaneously. PACE_CHARGE_CTRL switches to HIGH while PACING_REF_PWM becomes a percentage representation of p_xPaceAmp to the maximum voltage possible. Conversely to the previous state, all pin value changes in this state allow the current to flow to the primary capacitor to build up charge.

Since the discharging of capacitor C21 and the charging of C22 occur in parallel, these two substates may collectively be called the recharging or refractory state. However, for clarity and ease of troubleshooting, they are separated into two states.

These three substates then continue in a cycle until m_magnet changes such that the pacing is no longer XOO. View the latter portion of this section for the comprehensive stateflow diagrams. The direction of flow in these states is controlled by internal switches.

Table 2: XOO Programmable parameters

Parameter Name	**Function**	**Default Value**
p_xooInterval	Sets the interval between rising edges of aortic or ventricular pacing	1000 ms
p_xPaceWidth	Sets the pulse width of aortic or ventricular pacing	0.4 ms
p_xPaceAmp	Sets the magnitude of aortic or ventricular pacing discharge	3500 mV

p_xPaceAmp is used to control the amplitude of the voltage discharged to the atrium or ventricle in the XOO modes. It was initially defined with a value of 3500 mV as per the requirements, and has a tolerance of ± 12%. The decision to implement the amplitude in millivolts is due to the relatively small voltage values required in the pacemaker, allowing for more precise control of voltage delivery. It additionally circumvents floating-point errors that may occur when dividing decimal values.

p_xPaceWidth is used to control the duration of pacing delivered to the heart. It was implemented in the model in milliseconds to allow for greater precision and control. The default value for the pace width is 0.4 ms for XOO modes.

p_xooInterval controls the frequency of pulse delivery, representing the period between each pulse in milliseconds, which allows control over pacing. It is analogous to p_lowrateInterval in the XXI modes. Due to this, p_xooInterval and p_xPaceWidth then represent the refractory or recharging period between pacing.

It was decided to use p_xooInterval and p_xPaceWidth to implement the Periodic() function described in the requirements. This function controls the pacing state of the XOO mode by cycling from true to false. The function was then implemented such that it would start with a value of false, corresponding to a refractory state, and remain false for p_xooInterval-pxPaceWidth. It would then switch to true, corresponding to the pacing state, and remain in that state for p_xPaceWidth, before cycling back.

Controlling the pacing state this way ensures that pacing only occurs if there was no pacing during the previous time step, as the state cycles between true and false every time interval. This is critical to ensure pacing is delivered periodically, not constantly. Additionally, starting the XOO modes from the refractory state prevents pacing from discharging too quickly after switching into XOO mode from a previous mode.

2.3.1.2. Hardware Inputs and Outputs

The XOO modes operate without detection from hardware, as they are meant to mimic the natural, innate pacing of the heart. Signals are not monitored, and the pacing is instead determined by the parameters discussed in the previous section. However, hardware signals are controlled by the XOO modes.

Capacitor C22 is the primary capacitor, which provides the pacing of electricity into the atrium or ventricle in the XOO modes. The charging and discharging of this capacitor is thus the most critical component that must be controlled by the software. It is charged by the switches pertaining to pins D2, D5, D11, and D12 while being charged by the PWM signal. When discharged, the charge buildup in the primary capacitor then discharges into the heart and then into capacitor C21. The leading edge pulse in a pacing waveform is caused by the drainage of current from C22 into C21 and the heart.

Capacitor C21 is known as the blocking capacitor. It drains charge from the preceding capacitor C22, the primary capacitor, when pacing occurs. To ensure a net zero flow of current into the heart, the blocking capacitor C21 must also be drained through the heart based on switches changing from pins D10-D12. In

an ideal pacing waveform, the recharge pulse is caused by C21 generating that opposing current (with respect to C22's current) to bring the net current back to zero.

It is additionally critical to consider the order in which the pins are written to, to prevent direct connection of the PWM voltage to the heart. Thus, upon entry to a state which does not use a pacing pin, the pacing pin is disabled before any other pins' states are changed. This ensures that no unwanted voltage is delivered to the heart and prevents interference between hardware components.

Tables 3-6: Pin statuses for each of the states for the XOO modes. For the sake of conciseness, the substates which discharge C21 and charge C22 have been grouped together into one recharging substate.

State: Pacing Atrium		
Variable Name	**Corresponding Pin**	**Value**
PACE_CHARGE_CTRL	D2	LOW
PACE_GND_CTRL	D10	HIGH
VENT_PACE_CTRL	D9	LOW
VENT_GND_CTRL	D12	LOW
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
ATR_GND_CTRL	D11	LOW
ATR_PACE_CTRL	D8	HIGH

State: Recharging Atrium		
Variable Name	**Corresponding Pin**	**Value**
ATR_PACE_CTRL	D8	LOW
VENT_PACE_CTRL	D9	LOW
PACING_REF_PWM	D5	Duty cycle (1-100)
PACE_CHARGE_CTRL	D2	HIGH
PACE_GND_CTRL	D10	HIGH
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW

VENT_PACE_CTRL	D9	LOW
VENT_GND_CTRL	D12	LOW
ATR_GND_CTRL	D11	HIGH

State: Pacing Ventricle		
Variable Name	**Corresponding Pin**	**Value**
PACE_CHARGE_CTRL	D2	LOW
PACE_GND_CTRL	D10	HIGH
ATR_PACE_CTRL	D8	LOW
ATR_GND_CTRL	D11	LOW
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
VENT_GND_CTRL	D12	LOW
VENT_PACE_CTRL	D9	HIGH

State: Recharging Ventricle		
Variable Name	**Corresponding Pin**	**Value**
ATR_PACE_CTRL	D8	LOW
VENT_PACE_CTRL	D9	LOW
PACING_REF_PWM	D5	Duty cycle (1-100)
PACE_CHARGE_CTRL	D2	HIGH
PACE_GND_CTRL	D10	HIGH
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
ATR_GND_CTRL	D11	LOW
VENT_GND_CTRL	D12	HIGH

2.3.1.3. State Machine Design

As discussed earlier in this documentation, XOO pacing can be broken down into three substates: discharging the blocking capacitor, pacing the heart, and charging the primary capacitor.

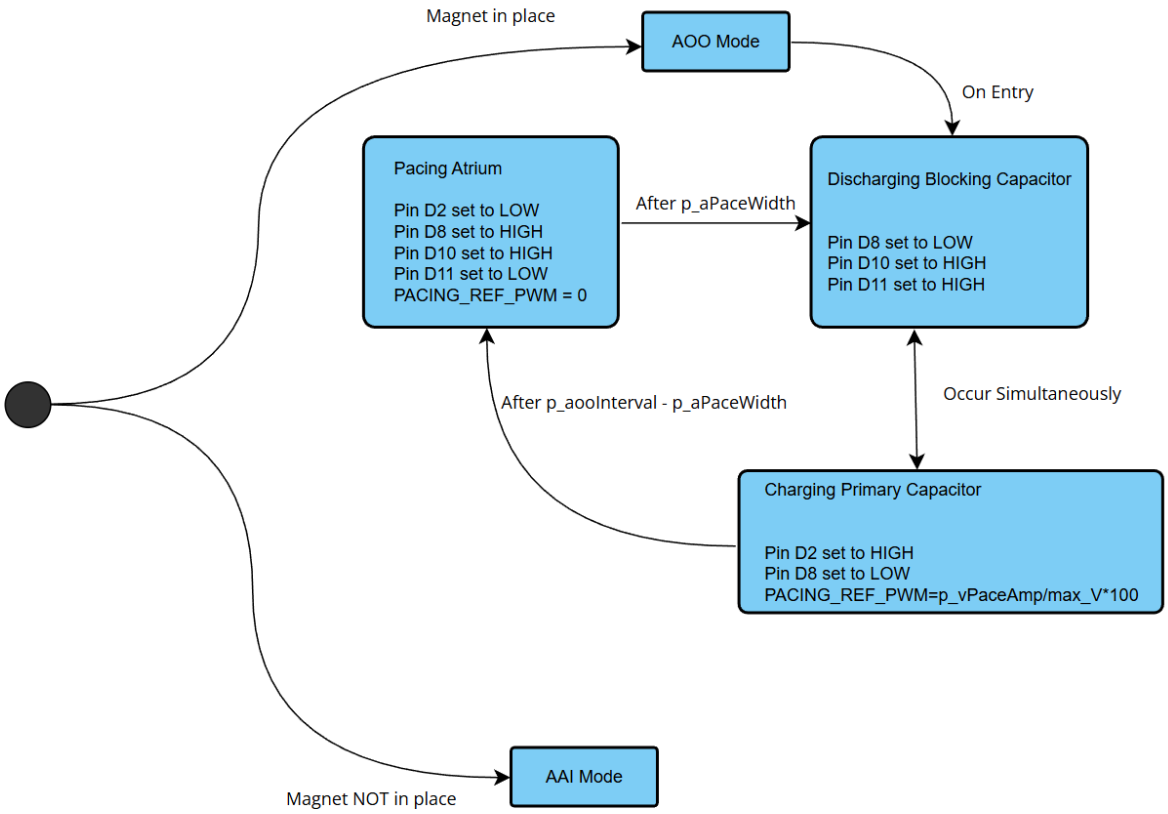

Figure 1. State Machine Diagram for AOO mode.

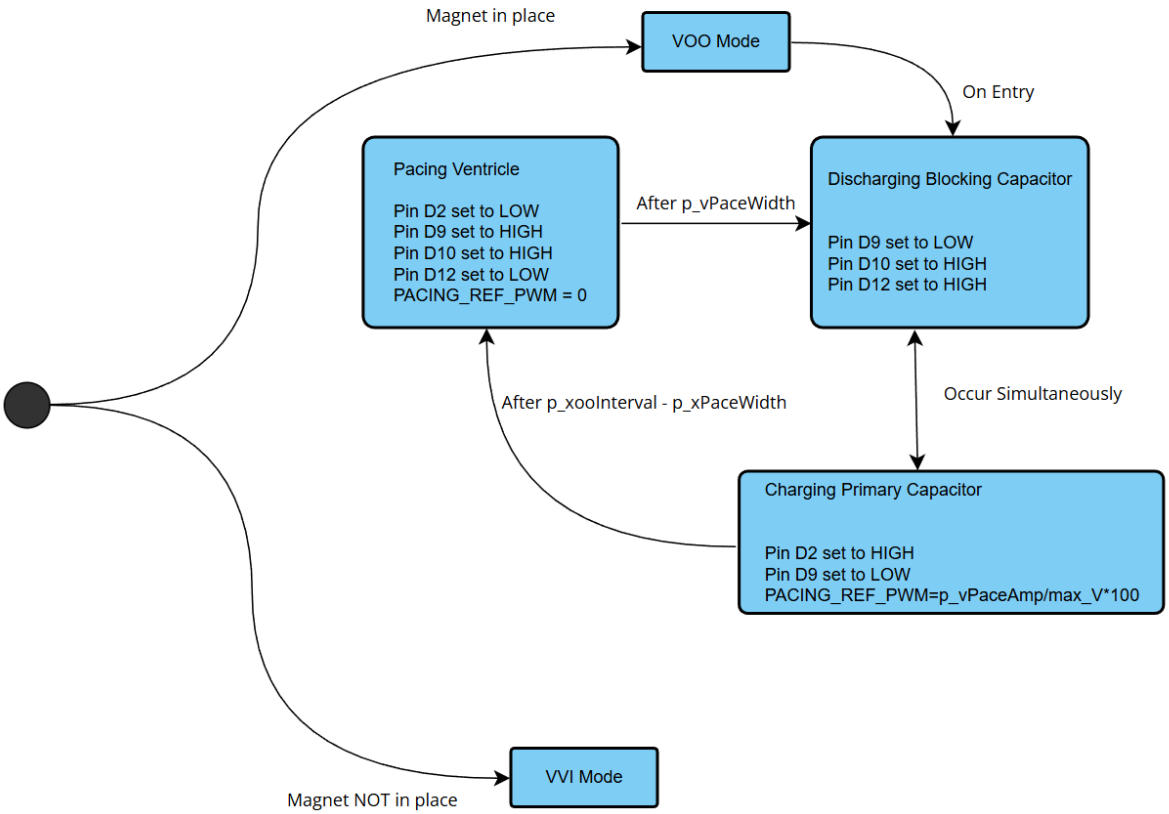

Figure 2. State Machine Diagram for VOO mode.

2.3.1.4. Simulink Design

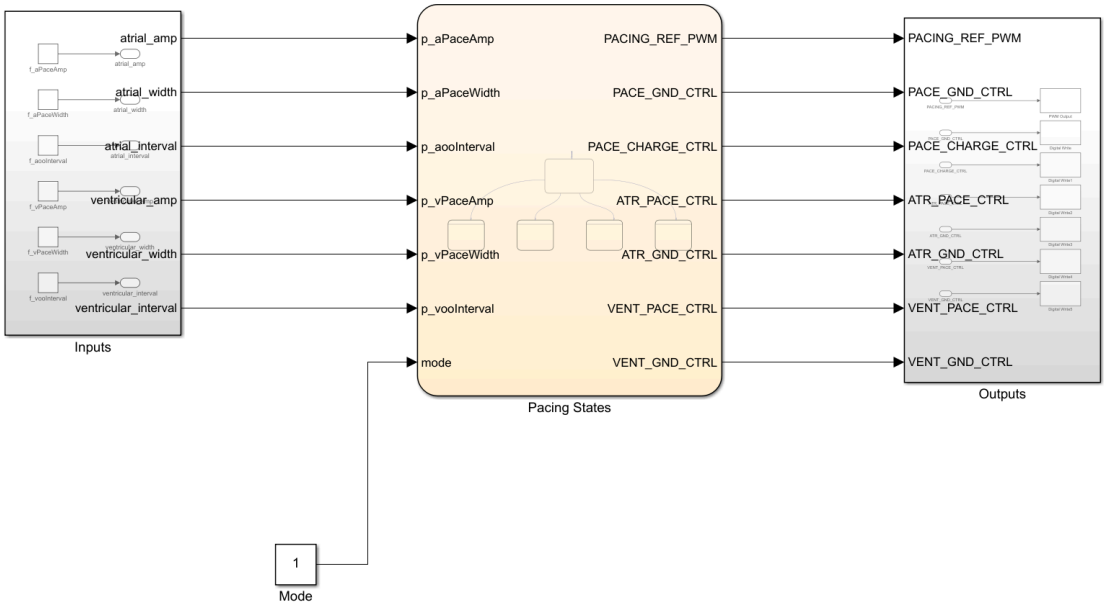

Figure 3. XOO Simulink Inputs and Outputs

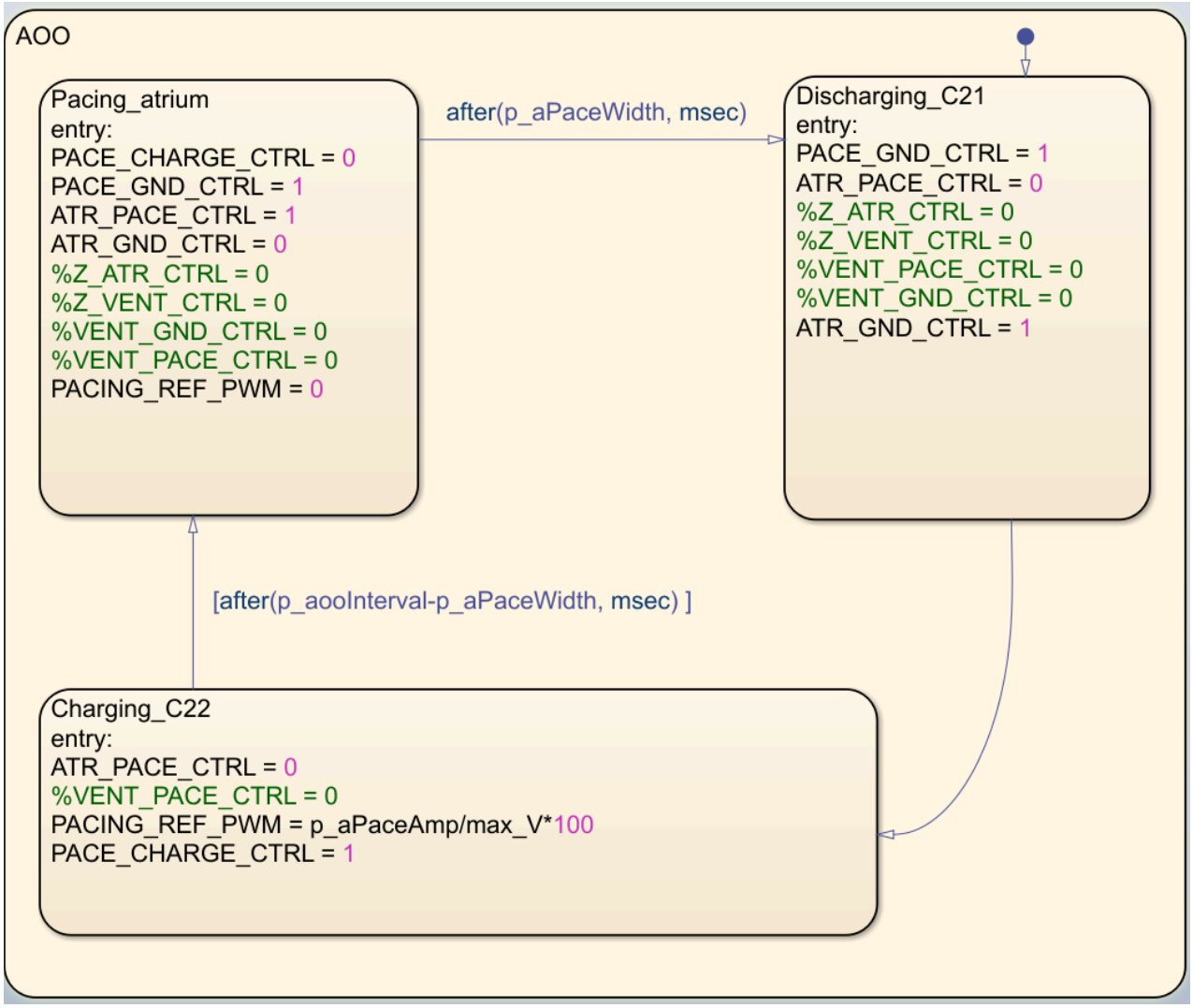

Figure 4. AOO Simulink Stateflow.

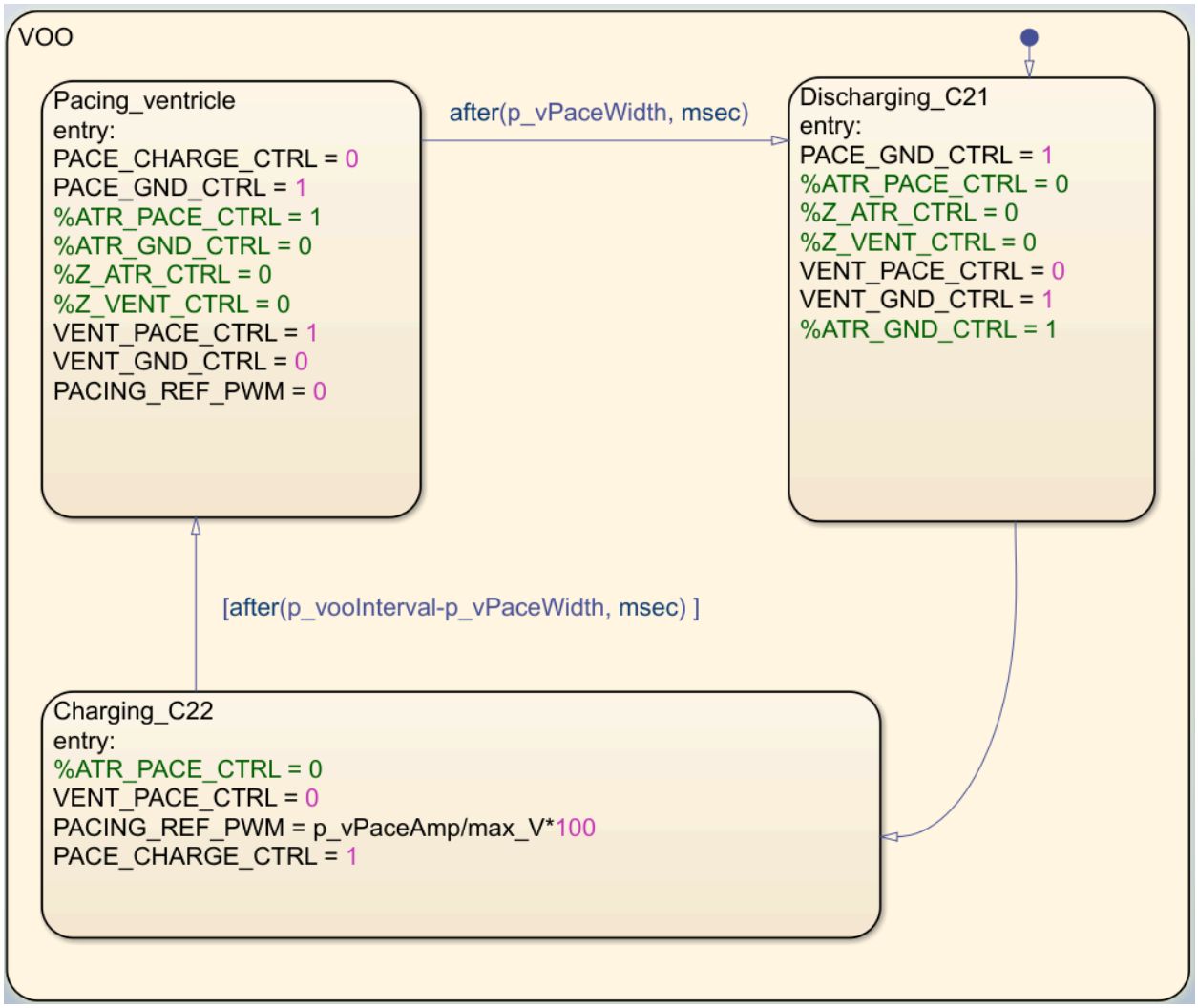

Figure 5. VOO Simulink Stateflow.

2.3.2 AAI and VVI Modes

2.3.2.1. System Architecture

The AAI and VVI implementations extend the AOO and VOO modes by adding intrinsic beat sensing and inhibition. These modes ensure the pacemaker only delivers a beat when the heart fails to beat on its own. It uses four main substates: charging the primary capacitor, pacing the heart chamber, discharging the blocking capacitor, and the refractory period. The sensing circuitry is active throughout the circuitry (FRONTEND_CTRL = HIGH so intrinsic atrial and ventricular activity can be sensed).

Charging of the primary capacitor C22 occurs when PACE_CHARGE_CTRL is set to HIGH. During this state, pacing outputs, x_PACE_CTRL, are set to LOW to prevent connecting the patient's atrium/ventricle directly to the PWM signal. The PACING_REF_PWM (PWM) is set to a percentage representation of p_xPaceAmp to the maximum voltage possible. If no intrinsic beat is sensed during the programmed

interval, the system will transition to the pacing state. If an intrinsic beat is sensed during the programmed interval, the system will transition directly to the refractory period.

During the pacing state, the device delivers a pulse to the atrium or ventricle depending on the mode. PACE_GND_CTRL and x_PACE_CTRL for the active pacing chamber (ATR_PACE_CTRL for atrial pacing and VENT_PACE_CTRL for ventricular pacing) are set to HIGH while PACE_CHARGE_CTRL and x_GND_CTRL are set to LOW. These pin values allow current to flow through the heart to deliver a pulse. The pulse is delivered at the programmed amplitude for the programmed duration before transitioning to the discharging state.

After pacing, the blocking capacitor C21 is discharged. During this state, both PACE_GND_CTRL and x_GND_CTRL for the active pacing chamber (ATR_GND_CTRL for atrial pacing and VENT_GND_CTRL for ventricular pacing) are set to HIGH while the pacing output x_PACE_CTRL is set to LOW. This ensures safe discharging of the current accumulated from pacing. The discharging state automatically transitions to the refractory period, without any condition or wait interval.

Following any paced or sensed beat, the device will enter the refractory period state. While sensing remains enabled in this state, sensed beats are ignored to prevent double beats. After the programmed refractory period, the device will transition back to the charging state.

These four substates will continue cyclically. There are two possible routes depending on if an intrinsic beat is sensed. If no intrinsic beast is sensed, the path is Charging → Pacing → Discharging → Refractory → Charging. If an intrinsic beat is sensed, the path is Charging → Refractory → Charging. If m_magnet changes (magnet is applied), the device will change modes to XOO.

Table 7: XXI Programmable parameters

Parameter Name	**Function**	**Default Value**
p_lowrateInterval	Sets the interval between rising edges of aortic/ventricular pacing	1000 ms
p_xPaceWidth	Sets the pulse width of aortic/ventricular pacing	0.4 ms
p_xPaceAmp	Sets the magnitude of aortic/ventricular pacing discharge	3500 mV
p_xRP	Sets the interval of the atrial/ventricular refractory period	320 ms

The programmable parameters p_xPaceAmp and p_xPaceWidth were implemented using the same reasoning and values as the XOO modes. These parameters define the pacing amplitude and pulse width respectively.

p_lowrateInterval represents the total pacing cycle length, which is the maximum duration from one paced pulse to the next if no intrinsic pulse is sensed. For XXI modes, this is 1000 ms, meaning the pacemaker will deliver a pulse every 1 second if no intrinsic beat is sensed.

The total pacing cycle (p_lowrateInterval) is divided into 3 phases: the pacing pulse, refractory period and waiting period. p_xPaceWidth represents the duration of the paced pulse and is set to a short value, 0.4 ms. The refractory period occurs after a paced or a sensed beat and is set to last 320 ms (p_xRP). The waiting period is the remainder of the p_xxiInterval after p_xPaceWidth and p_xRP are subtracted. It will wait this waiting period for a natural beat, charging the capacitor to prepare for a paced pulse if no natural beat is sensed.

2.3.2.2. Hardware Inputs and Outputs

The XXI modes expand upon the XOO modes by implementing intrinsic beat sensing and inhibition. Unlike XOO modes which deliver pulses at fixed intervals, the XXI modes monitor natural heart signals to determine whether an intrinsic beat occurs before delivering a pulse. Based on feedback from the hardware, the software transitions between sensing, pacing and refractory states. However, output of the hardware pins are controlled by the XXI modes through software to determine when pacing should occur.

Capacitor C22 remains the primary capacitor and provides the pacing of electricity into the atrium or ventricle in the XXI modes. It is charged by the switches controlled by pins D8, D9, D5 and D2 using the PWM signal. While charging, the system monitors for sensed intrinsic beats. If no intrinsic beat is sensed within the programmed interval, the capacitor C22 discharges into the heart and then into capacitor C21 in order to deliver a pulse. The leading edge pulse in a pacing waveform is caused by the drainage of current from C22 into C21 and the heart.

Capacitor C21, the blocking capacitor, drains charge from capacitor C22 when pacing occurs. After the pacing pulse, the blocking capacitor C21 is also discharged through the heart to ensure a net zero flow of current through the heart. In an ideal pacing waveform, this discharge is represented by the recharge pulse (opposing current with respect to C22's current flows back through the heart to bring current back to zero). This discharging process occurs through switches controlled by pins D10-D12.

In the XXI modes, sensing and inhibition also occurs. The sensing circuitry is enabled by the pin D13 (FRONTEND_CTRL) which is always set to HIGH to allow for ventricular and atrial sensing. These modes use comparator PWM signals from pins D3 (VENT_CMP_REF_PWM) and D6 (ATR_CMP_REF_PWM) which establish a threshold voltage for beat detection.

The actual beat detection occurs through pins D0 (ATR_CMP_DETECT) and D1 (VENT_CMP_DETECT), which output a HIGH signal when the sensed cardiac voltage is greater than the threshold voltage and outputs LOW otherwise. These signals become inputs for the Stateflow chart as m_vs and m_as, allowing the system to respond appropriately (either deliver a pulse or inhibit based on intrinsic beats).

To determine an appropriate threshold value for x_CMP_REF_PWM, the natural heart pacing can be observed by connecting A1 (VENT_SIGNAL) for VVI and A0 (ATR_SIGNAL) for AAI to a scope. Then, an approximate threshold can be determined from the rectified output which is about 0.52V as seen from the scope graphs below for the VVI and AAI modes respectively. There are gaps in the scope graphs, however, the general idea can still be seen. This comparator can also be tested by directly connecting D1/D0 to RED_LED and observing if the red led flashes at the same time as the ATR or VENT lights on the heart circuit board.

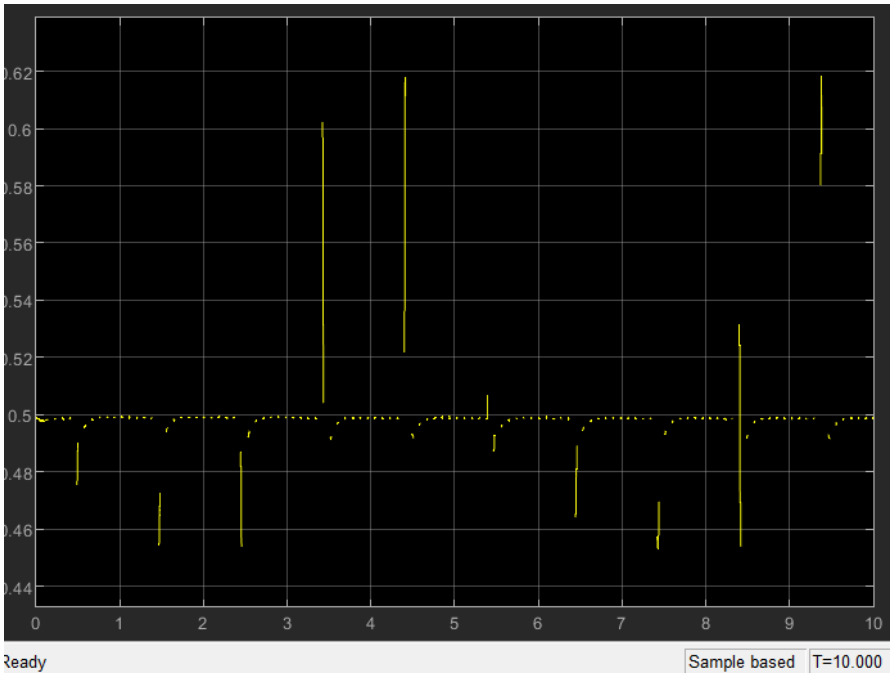

Figure 6. Scope readings of ventricular (A1) signals showing the natural pacing waveform used to determine the 0.52 V threshold voltage for VENT_CMP_REF_PWM.

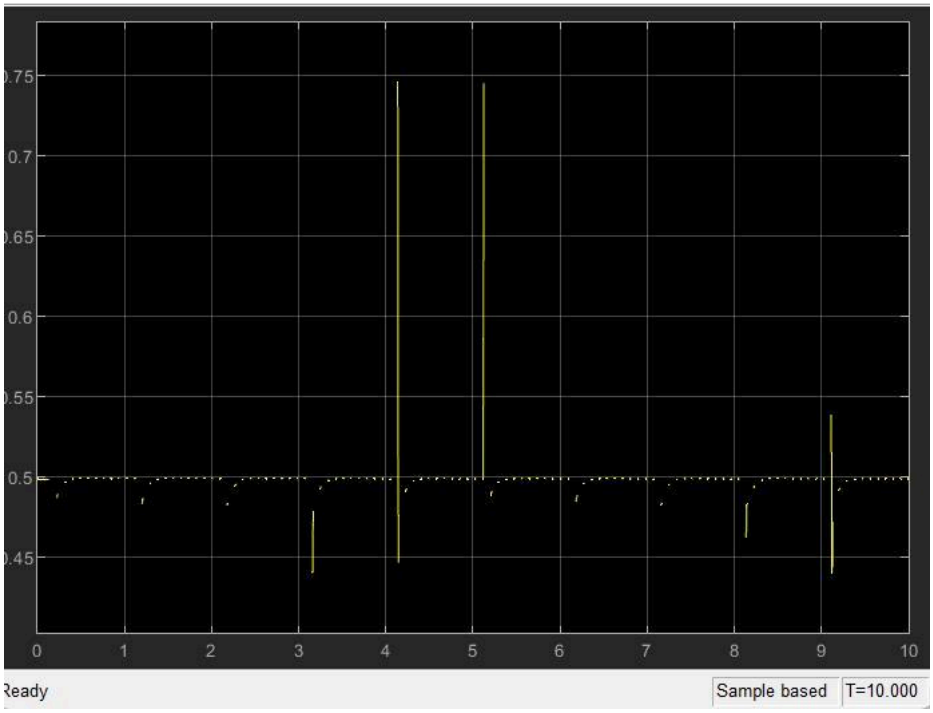

Figure 7. Scope readings of atrial (A0) signals showing the natural pacing waveform used to determine the 0.52 V threshold voltage for ATR_CMP_REF_PWM.

Tables 8-13: Pin statuses for each of the states for the XXI modes, as well as the non-state specific pins. For the sake of conciseness, the substates which discharge C21 and charge C22 have been grouped together into one recharging substate.

Sensing and Threshold Control Pins (Atrial)		
Variable Name	**Corresponding Pin**	**Value**
FRONTEND_CTRL	D13	HIGH
ATR_CMP_DETECT	D0	HIGH/LOW
ATR_CMP_REF_PWM	D6	52

State: Pacing Atrium		
Variable Name	**Corresponding Pin**	**Value**
PACE_CHARGE_CTRL	D2	LOW
PACE_GND_CTRL	D10	HIGH
ATR_PACE_CTRL	D8	HIGH
ATR_GND_CTRL	D11	LOW

Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
VENT_GND_CTRL	D12	LOW
VENT_PACE_CTRL	D9	LOW

State: Recharging Atrium		
Variable Name	**Corresponding Pin**	**Value**
ATR_PACE_CTRL	D8	LOW
VENT_PACE_CTRL	D9	LOW
PACING_REF_PWM	D5	Duty Cycle (1-100)
PACE_CHARGE_CTRL	D2	HIGH
PACE_GND_CTRL	D10	HIGH
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
VENT_GND_CTRL	D12	LOW
ATR_GND_CTRL	D11	HIGH

Sensing and Threshold Control Pins (Ventricular)		
Variable Name	**Corresponding Pin**	**Value**
FRONTEND_CTRL	D13	HIGH
VENT_CMP_DETECT	D1	HIGH/LOW
VENT_CMP_REF_PWM	D3	52

State: Pacing Ventricle		
Variable Name	**Corresponding Pin**	**Value**
PACE_CHARGE_CTRL	D2	LOW
PACE_GND_CTRL	D10	HIGH

ATR_PACE_CTRL	D9	LOW
ATR_GND_CTRL	D12	LOW
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
VENT_GND_CTRL	D11	LOW
VENT_PACE_CTRL	D8	HIGH

State: Recharging Ventricle		
Variable Name	**Corresponding Pin**	**Value**
ATR_PACE_CTRL	D8	LOW
VENT_PACE_CTRL	D9	LOW
PACING_REF_PWM	D5	Duty Cycle (1-100)
PACE_CHARGE_CTRL	D2	HIGH
PACE_GND_CTRL	D10	HIGH
Z_ATR_CTRL	D4	LOW
Z_VENT_CTRL	D7	LOW
ATR_PACE_CTRL	D9	LOW
ATR_GND_CTRL	D12	LOW
VENT_GND_CTRL	D11	HIGH

2.3.2.3. State Machine Design

As discussed earlier in this documentation, XXI pacing can be broken down into four substates: discharging the blocking capacitor, the refractory period, pacing the heart, and charging the primary capacitor.

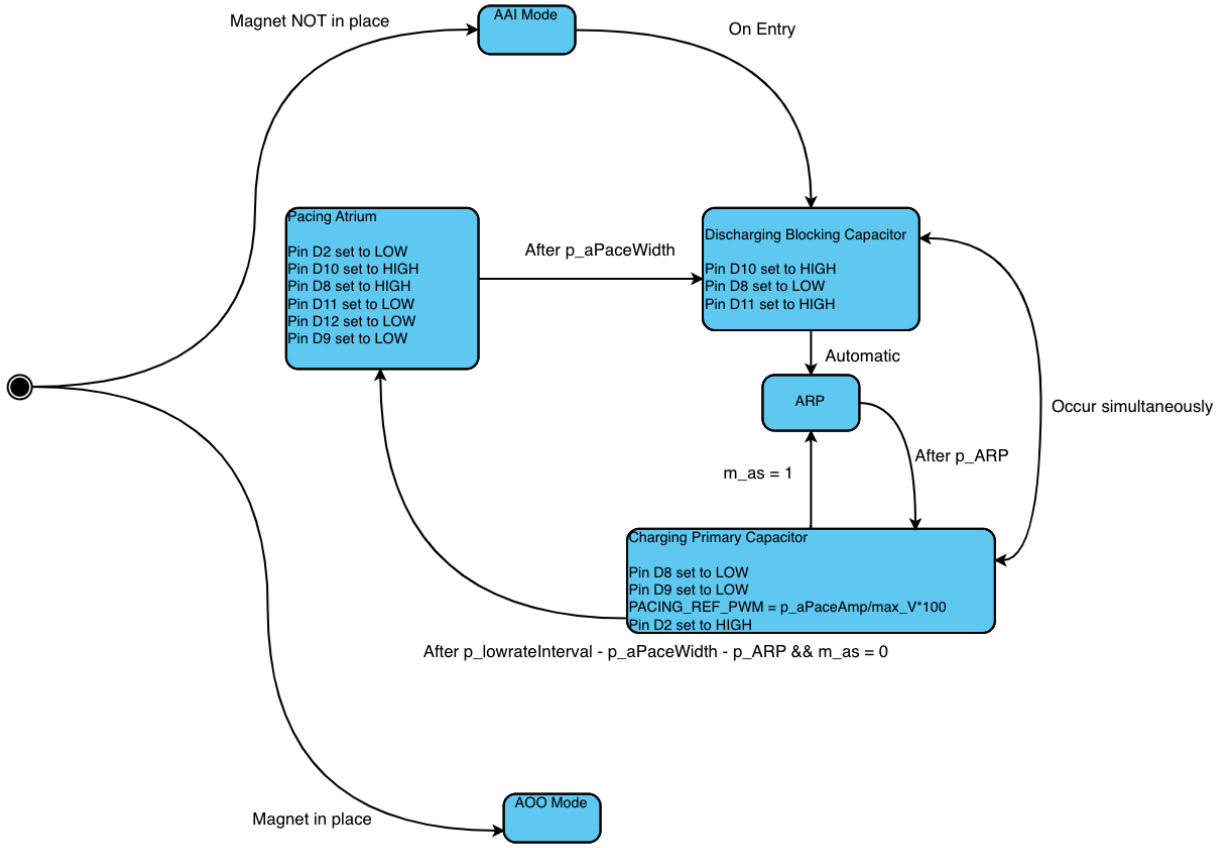

Figure 8. State Machine Diagram for AAI mode.

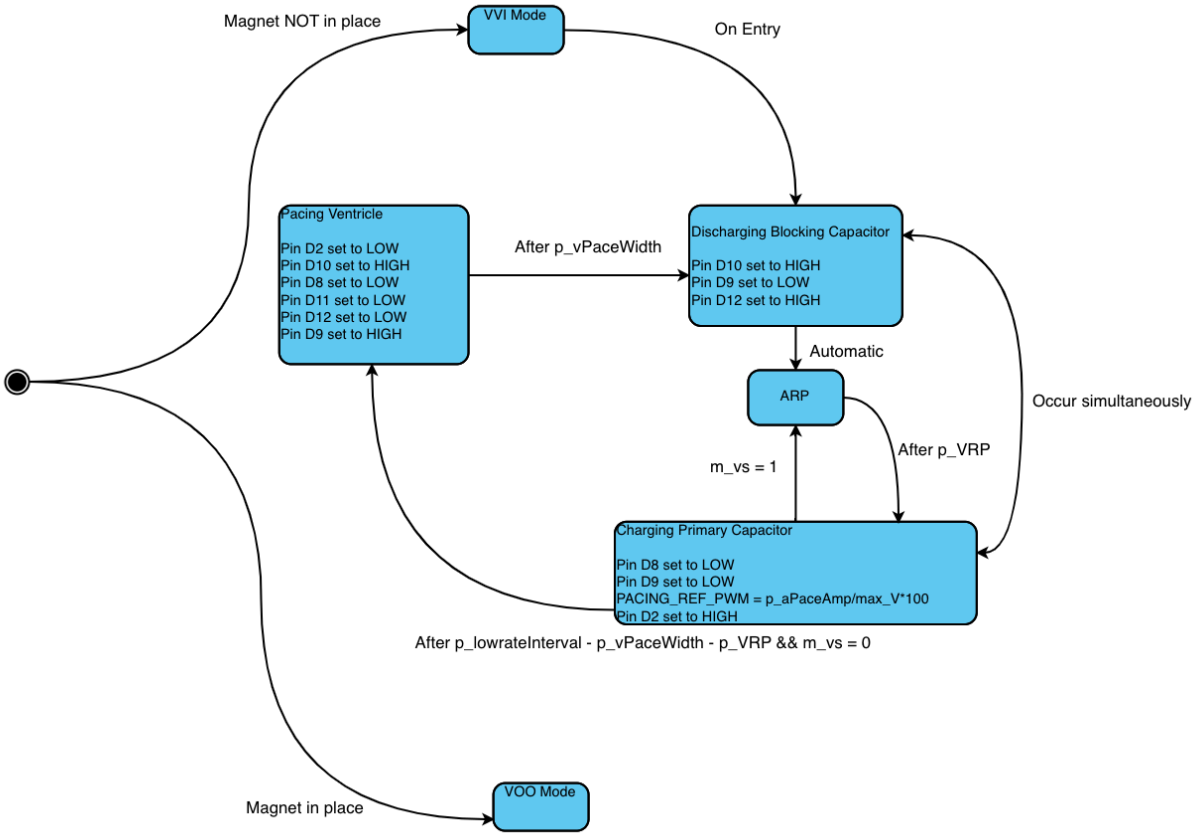

Figure 9. State Machine Diagram for VVI mode

2.3.2.4. Simulink Design:

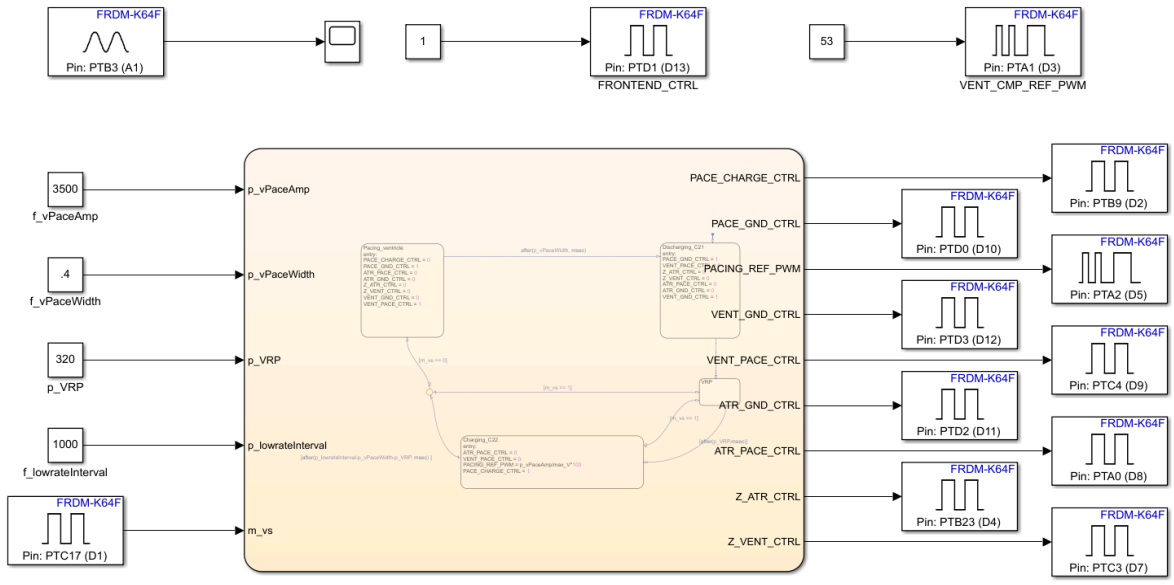

Figure 10. VVI Simulink Inputs/Outputs

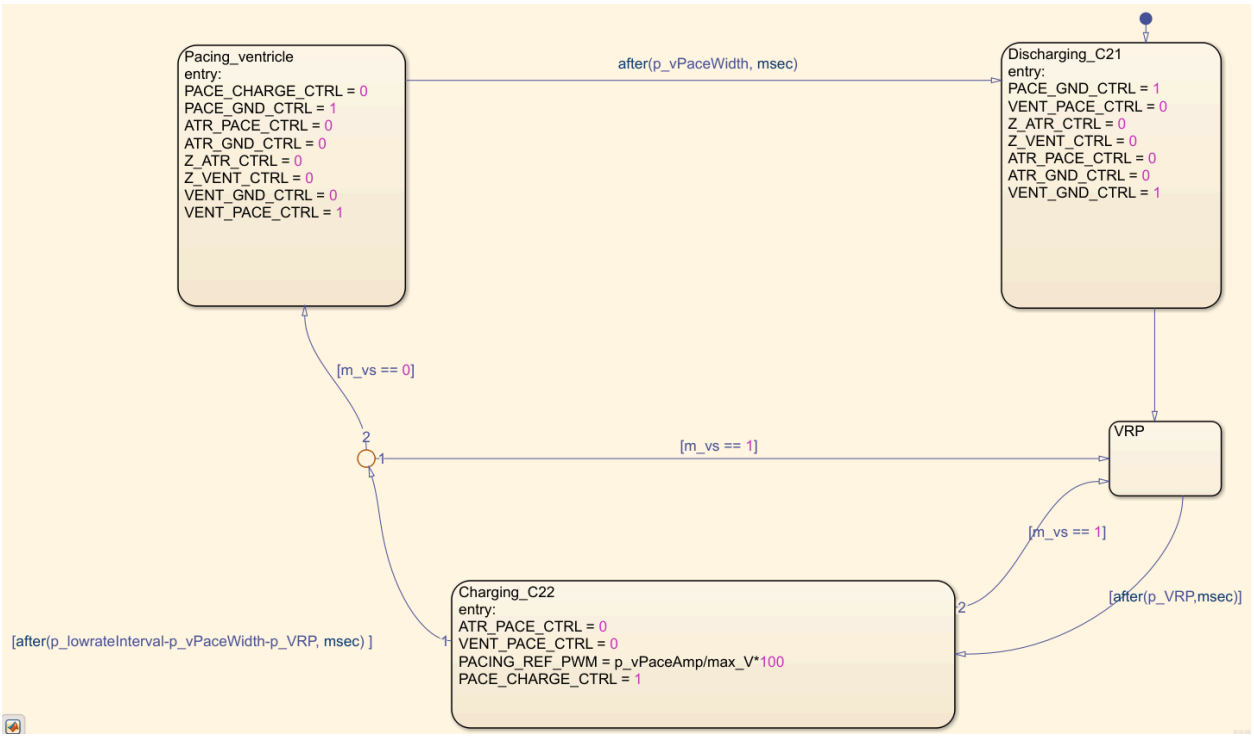

Figure 11. VVI Simulink Stateflow

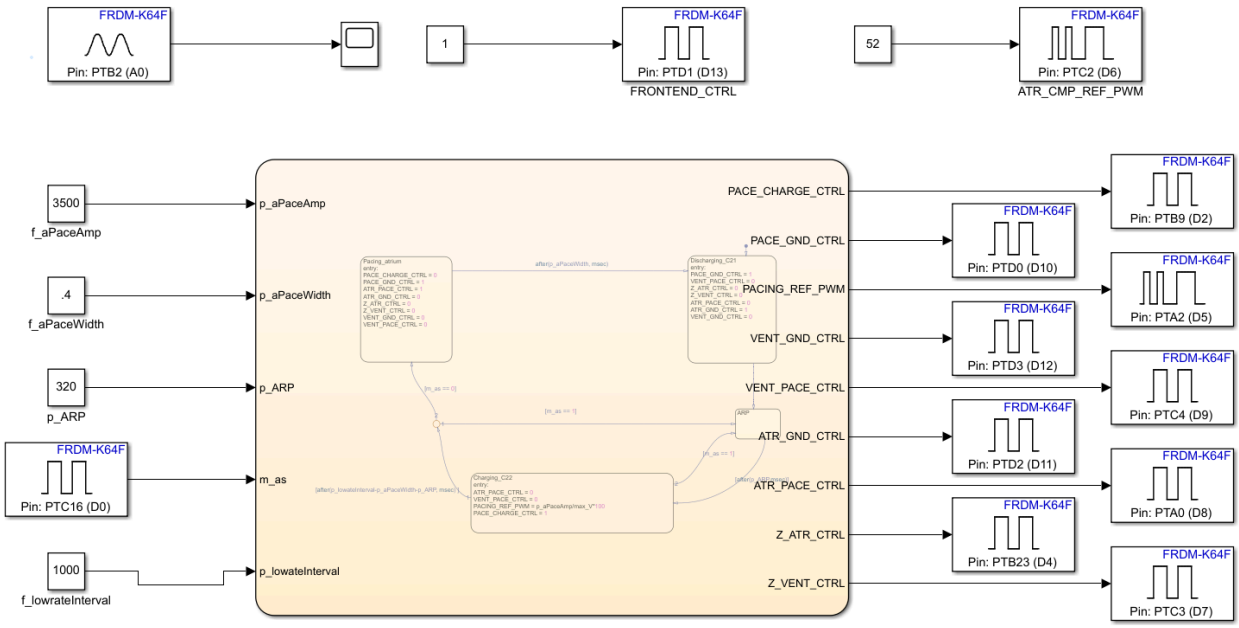

Figure 12. AAI Simulink Inputs/Outputs

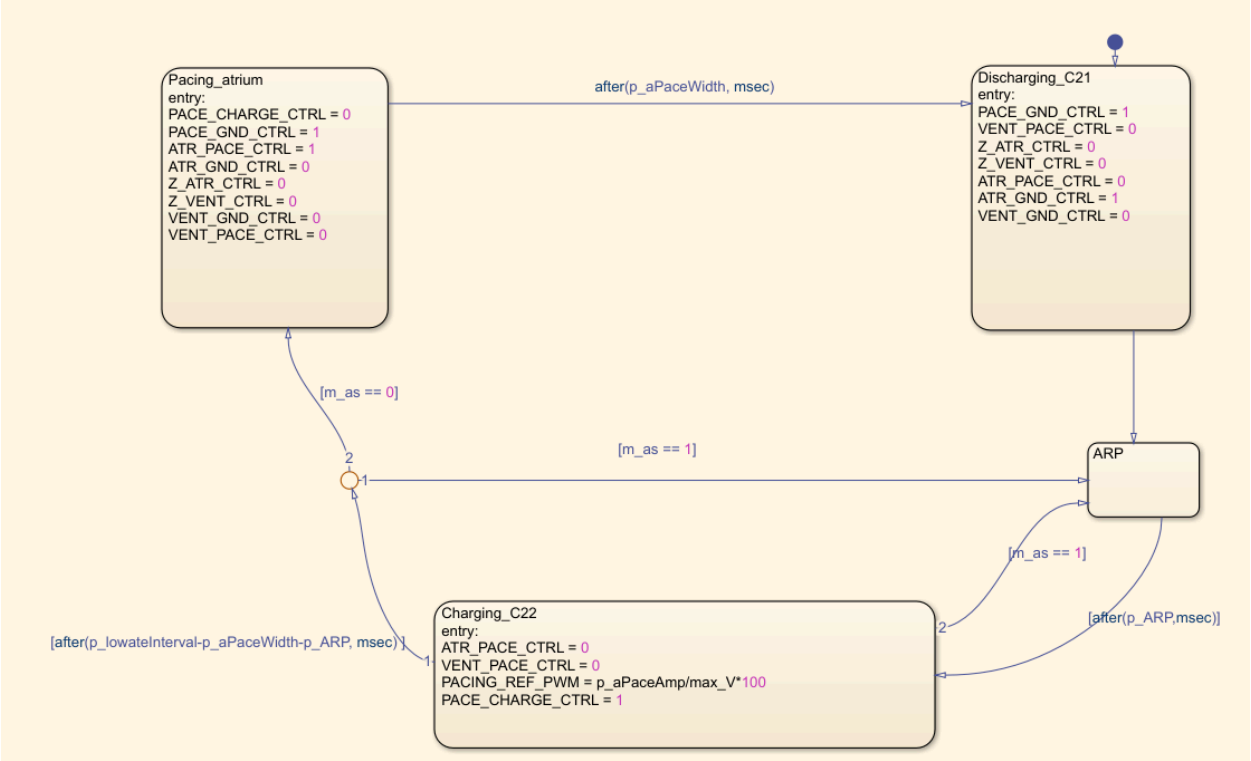

Figure 13. AAI Simulink Stateflow

2.3.3. DCM

2.3.3.1. System Architecture

The DCM follows a modular GUI architecture built using PyQt5 - a python framework used for creating applications. It has an expansive library of widgets and allows for easy event-driven programming. The application is written in Python, a commonly used language for creating GUI's. It can also help with future compatibility with hardware.

The DCM uses a stacked navigation model implemented through QStackedWidget, which organizes the main application windows: LoginPage, DashboardPage, and ModeEditorPage. The structure provides linear navigation flow while preserving the application state between page transitions. Each page is encapsulated in its own class to improve cohesion and organization.

At the top level, the MainWindow class acts as the main controller that contains the stacked widget, manages users, stores hardware states and provides access to global menus that are visible across all pages. The top bar includes the File menu, which allows the user to export parameter data and reports, the Simulator menu, which allows for changes in the status of communication, telemetry and hardware, which is sufficient for Deliverable 1, and the Utilities menu, which includes pop-up About and Set Clock windows for additional information and time configuration.

Navigation begins at the LoginPage, where users can register or login, given a maximum capacity of 10 users. Passwords are hashed using SHA-256 before storage to ensure secure authentication. SHA-256 hashing was used for passwords to ensure unique username and password combinations for each user, while also making sure the data is stored securely. Storing raw passwords can be dangerous. In a safety-critical system such as a pacemaker, any and all opportunities for an unauthorized person to access a user's information must be limited.

The Database class is crucial for encapsulating file input/output, reading and writing a JSON file (dcm_d1_dile.json). JSON was selected because it provides a human-readable format that supports structured storage without requiring external databases. This is ideal for storing usernames, passwords, and parameters and makes it easier to understand.

After logging in, users are taken to the DashboardPage, which is where they can click a button to choose their desired mode, directing them to their respective ModeEditorPage. The default tab upon entry is the Parameter tab, where parameters are stored and can be modified within a set limit. Validation was implemented through a validate method using QValidator to only allow specific values to be inputted in the parameters, based on the provided information. A stepBy function was also added to clamp onto the closest allowed parameter value, given one that isn't following the specified step value within the range.

There are two other tabs present at the top. The Summary tab shows the Summary of the mode that is displayed, including their parameters, and is then graphically shown in Egram (D2). The Egram tab was done using QPainter, QPen and QTimer. A PNG of this graph can be saved onto the user's device.

2.3.3.2. Choice of Imports and Libraries

The application uses PyQt5 as the desktop GUI due its diverse toolbox of layouts, widgets, dialogs, menus, timers, and graph rendering for Egram using QPainter, QPen and QColor without external plotting. QTimer is used as a time-based update to animate Egram in real time. The PyQt5.QtGui imports of QIcon, QValidator, and QTextDocument are used for the visual components of the GUI. QIcon provides recognizable symbols or icons across buttons and menus to improve clarity. QValidator offers inline input validation for text based widgets which restricts users from inputting invalid data. And QTextDocument is used for formatting parameter reports into printable PDF files without requiring additional external libraries.

The widgets under PyQt5.QtWidgets are used to establish basic functionality of the DCM as listed below.

The QApplication is used to start the Qt event loop allowing for a single process-wide object that manages all GUI events within the singular DCM application.

QWidget is used as a base class for most UI components, offering containers for custom pages or sections with minimal resources. Allows for modular development for each page to be an independent widget.

QMainWindow provides a desktop frame with a menu bar, status bar, and central widget, ideal for an all encompassing interface controller.

QStackedWidget operates similarly to a "deck" that allows the user to switch between pages, allowing for an intuitive multi-page flow while preserving the state of each page.

QVBoxLayout and QHBoxLayout are used as the vertical and horizontal layout manager, offering consistent spacing, resizing, and eliminates manual math.

QLabel is used for simple or rich text for titles, hints, and live statuses within the pages, necessary for viewability of information within the GUI.

QLineEdit is a single-line text input allowing for user inputs, increasing interactability between the frontend and backend of the software.

QPushButton is a clickable button that calls upon an action, acting as an intuitive trigger for the user with clear boundary indications for the button.

QMessageBox offers a modal dialog that can provide information, warning, or error messages to the user. It offers consistent UX without requiring custom dialogs.

QFormLayout offers 2-column “Label:Field” grids for parameter entry, making it easier to read and create the layout and enhancing UX.

QSpinBox uses integer only inputs, switching between a built in range or step control, but overrideable. Useful for a range of values with different step values and can also be typed in.

QDoubleSpinBox uses floating-point only inputs, switching between fixed decimals and supports rounding or format control. It can be used to prevent float drifting. Has similar features as QSpinbox and allows users to step up or step down in value, either with the arrows or via typing. Useful for the float step increments.

QComboBox offers a drop-down selection list and is useful for enumerations or selective values. Used for Mode, Hysteresis, Rate smoothing, toggling dependent widgets with a signal-slot method. The toggling of different widgets ensures only the valid data are utilized by the application.

QGroupBox groups related parameters, such as atrial and ventricular, with a box around a group of widgets and titles them. This enhances user readability.

QTabWidget encapsulates the tabs, such as the parameters, summary, and egram, in one place and reduces navigation while keeping the related views synchronized. This allows an intuitive viewing experience for users and allows for data retention between navigated pages.

QFileDialog uses a file picker dialog and is a native file picker for JSON exports and report or save actions, increasing cross-platform compatibility.

QAction is a menu item that can be clicked, used by the File, Simulator, and Utilities menu. The same QAction can be attached to menus across the pages to avoid redundancies and ensures consistent states across pages.

QActionGroup groups the actions together and can be used in a radio-button behaviour. This is useful for guaranteeing one active condition, used for the telemetry simulation conditions.

QStatusBar is a one-line bar at the bottom for the status bar that remains persistent throughout the entire interface, increasing users' awareness of the status for between the communication of the device, the telemetry conditions and the synchronized clock.

QDialog is a top-level window that provides brief or short-term communications to the user with modal interactions of About/Set Clock that allows you to return to the main window after the dialog is closed.

QDialogButtonBox allows for the user to interact with the dialog while presented in a suitable widget layout style.

QDateTimeEdit is used for structured date time input edits within the Set Clock dialog widget. It prevents invalid formats from being inputted and simplifies data stored as QDateTime.

The json library import is used to read/write JSON files, compatible with most desktops due to its lightweight nature and portable persistence.

The os library import is used to file paths and existence checks following the desktop's environment and path names.

The hashlib provides the SHA-256 password hashing so credentials are never stored in plain texts, increasing security for the stored credentials.

The typing annotation document imports Dict, Any, List, Optional which makes documenting and maintenance safer, also improving readability and maintenance with type hints.

The QIcon, QValidator, QFileDialog, and QStatusBar improves the user experience by adding icons, inline validation, and consistent file and save interactions. This creates a single file, zero-server and cross-platform app that can be run with just Python and PyQt5 installed.

2.3.3.3. Programmable Parameters

The ModeEditorPage deals with the programmable parameters that depend on provided ranges and the selected mode. In the DCM code, these parameters are defined as global constants that allow for easy referencing and ensure that these values cannot be changed by external influences. This is necessary for a safety-critical system where the allied values are key to ensuring a patient's health.

Lower Rate Limit (LRL) uses a piecewise step segment which varies the increments depending on the range it falls under (30-50 ppm uses 5-ppm steps, 50-90 ppm uses 1-ppm steps, and 90-175 ppm uses 5-ppm steps). An array is used to represent the maximum, minimum and step for each segment of the piecewise function. This segmentation provides flexibility for future parameter range updates.

Upper Rate Limit (URL) can more easily use uniform increments, as it relies on one set range, rather than a piecewise function like LRL does (the values are 50-175 ppm using 5-ppm steps). Since it does not require the piecewise step, specifying a maximum, minimum, and step as individual variables decreases the complexity compared to an array.

Pulse Width (PW) is similar to LRL, utilizing a piecewise step segment with floating-point values instead of integers (0.05, 0.1-1.9 ms with 0.1 ms steps). The design is similar to LRL, with values rounded to two decimal places to minimize floating-point errors and ensure the displayed and stored values remain consistent.

Amplitude is separated into atrial (0.5-3.2 V, 0.1 V steps) and ventricular (3.5-7.0 V, 0.5 V steps) ranges that support the different pacing configurations while maintaining simple, uniform step increments. They have separate minimums, maximums, and steps, and can be turned "Off" or "On" depending on the mode. The modular design decreases code complexity and simplifies parameters when used in the validation process.

Refractory periods are defined in an integer range with uniform step sizes. Ventricular and atrial modes use the same range and step (150-500 ms, 10 ms step). The integer type can be used based on the provided range.

Enumerations for MODES, HYSTERESIS_STATES, and RATE_SMOOTH_CHOICES are used in conjunction with the QComboBox widget. The combo box emits a selection change signal that connects to the slot methods in the ModeEditorPage, which toggles the relevant parameters. This design ensures that only the relevant parameters for each mode are enabled, preventing conflicting data entries.

App information data is a constant global variable, allowing the metadata to be stored and remain consistent throughout the entire interface. This ensures consistent representation of the software version and ownership in the About dialog and across all user interactive components.

The allowed list of global constants are used by the widgets that are dynamically generated by the helper function. This derives the global constants for the parameters into dynamically generated lists that are within the requirement's limits, prevents duplicates, and retains the decimal places for floating-point values. This design avoids hardcoded limits, allowing updates to automatically influence the rest of the application if parameter definitions are changed.

Table 14: DCM programmable parameters

Parameter	Widget / variable / class	Default value	Software Function
Pacing Mode	Variable: mode Widget: QComboBox	AOO / VOO/ AAI / VVI	Can switch between the different modes with a dropdown box
Lower Rate Limit	Class: LRLSpinBox Widget: QSpinBox	60 ppm	Determines the lowest rate limit LRL<URL with arrow increments, steps based on requirements
Upper Rate Limit	Class: URLSpinBox Widget: QSpinBox	120 ppm	Determines the highest rate limit URL>LRL with arrow increments, steps based on requirements
Atrial Amplitude	Class: AmplitudeWidget Widget: QDoubleSpinBox and QWidget	"On" (AOO,AAI) or "Off" (VOO, VVI), 3.0 V	Arrow fixed doubles step increments based on requirements only editable on the "On" state.
Ventricular Amplitude	Class: AmplitudeWidget Widget: QDoubleSpinBox and QWidget	"On" (VOO,VVI) or "Off" (AOO, AAI), 3.0 V	Arrow fixed doubles step increments based on requirements only editable on the "On" state.
Atrial Pulse Width	Class: PWSpinBox Widget: QDoubleSpinBox	0.40 ms	Arrow fixed doubles step increments based on requirements
Ventricular Pulse Width	Class: PWSpinBox Widget: QDoubleSpinBox	0.40 ms	Arrow fixed doubles step increments based on

			requirements
Atrial Refractory Period	Class: RefPeriodSpinBox Widget: QSpinBox	250 ms	Arrow fixed integer step increments based on requirements
Ventricular Refractory Period	Class: RefPeriodSpinBox Widget: QSpinBox	320 ms	Arrow fixed integer step increments based on requirements
Hysteresis	Variable: HYSTERESIS_STATES Widget: QComboBox	Off	Only available when pacing mode is set to AAI or VVI, blocked using QComboBox
Hysteresis Rate Limit	Widget: QComboBox	60 ppm	Only available when pacing mode is set to AAI or VVI, and Hysteresis is “ON” blocked using QComboBox
Rate Smoothing Up and Down	Variable: RATE_SMOOTH_CHOICES Widget: QComboBox	Off	Only available when pacing mode is set to AAI or VVI, blocked using QComboBox

2.3.3.4. Hardware Input and Output

Currently, the GUI is separated from the Simulink and hardware, but the necessary modules, classes and pages, such as the Egram (D2) page, have been created as placeholders to be changed and used for the next deliverable. Currently, the values, graphs and connection statuses are simulated. Future inputs from hardware will include sensed atrial and ventricular events, as well as potential surface data. More detailed information regarding device identity and link state may be included. Future outputs to hardware will include parameter sets. This can be achieved through another Qt feature, such as QSerialPort, that emits signals to MainWindow. The inclusion of messages for DCM and pacemaker communication can help with debugging and support hardware hiding - the GUI should not know if it’s interacting with a real pacemaker or a simulator.

The DCM does not deal with the physical pacemaker pins, but focuses on the interaction and communication between the pacemaker information and the code.

The communication between hardware and software with the GUI will be implemented in Deliverable 2.

2.3.3.5. State Machine Design for Each Mode

This information pertains to the mode design, not the DCM.

2.3.3.6. Simulink Diagram

As of now, the DCM does not have any connection to Simulink models or diagrams.

2.3.3.7. DCM Screenshots with Design Explanations

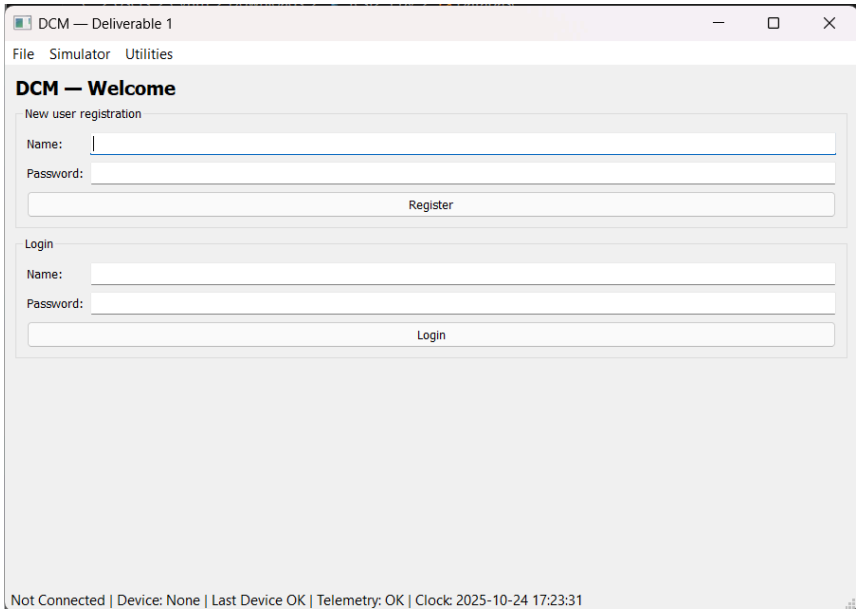

Figure 14: DCM LoginPage with login and registration fields

The LoginPage is designed with PyQt5 GUI with an intuitive interface that clearly separates registration and login sections to minimize user confusion. Named and Editable fields for the Name and Password with distinctly named Register and Login buttons enhances usability and simplifies user interaction.The status bar is used to provide real-time feedback for enhancing user's awareness of the current device connection and clock synchronization. This page will open first, designed to ensure authentication is required before access, maintaining security and privacy of user data.

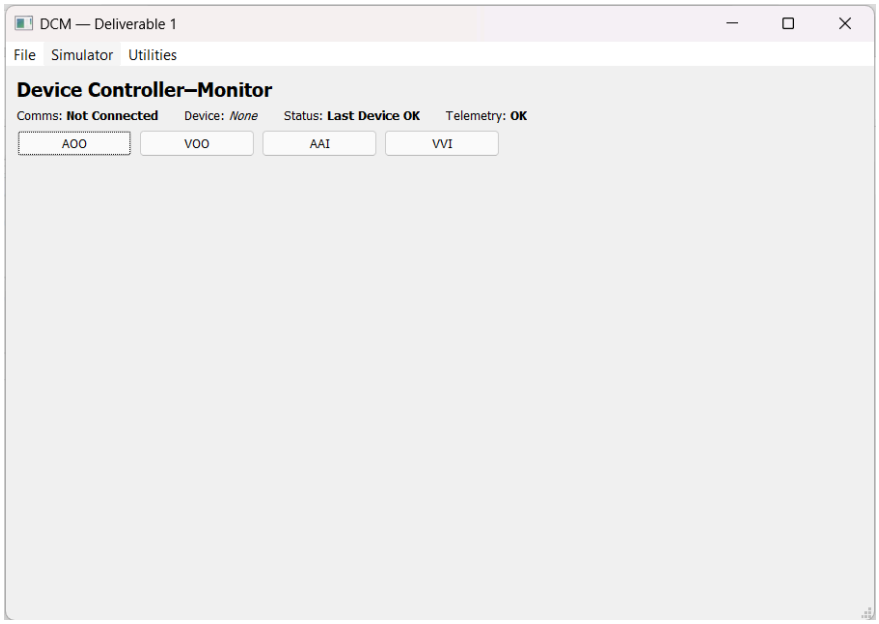

Figure 15: DCM DashBoardPage with status windows and pacing modes

The DashboardPage uses a user-friendly interface with labeled actionable buttons and statuses for mode interaction and system monitoring. The design is clear and ensures the user can identify and select the desired mode. The status indicator displays communication between hardware devices and performances. This layout aligns with modularity and enhances user awareness and understanding in a well organized interface.

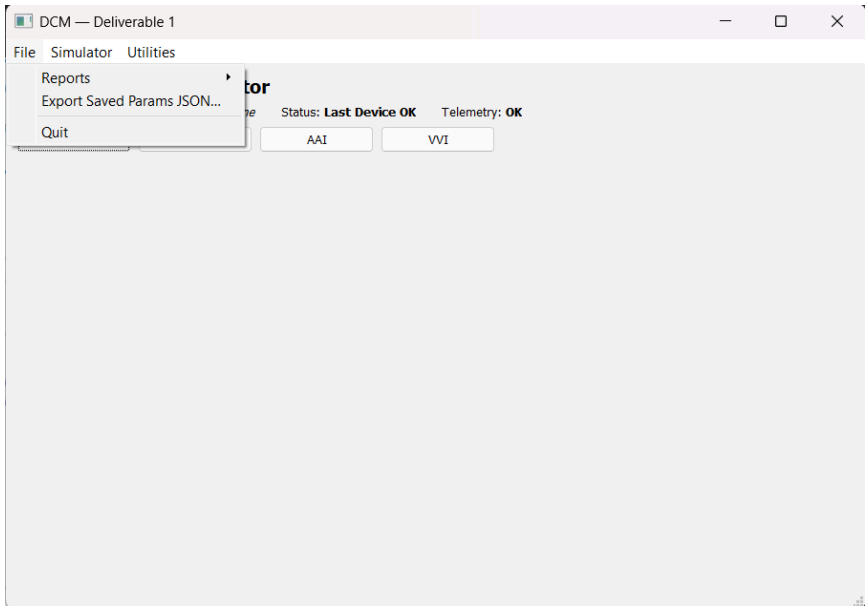

Figure 16: DCM DashBoardPage showcasing the File menu dropdown

File is a dropdown designed to provide essential file management functions for DCM, promoting usability and modularity. It allows users to generate documentation and save programmable parameters for future

use. The design as a dropdown menu allows for easy accessibility across pages without having to open a new window, improving workflow efficiency and an intuitive interface.

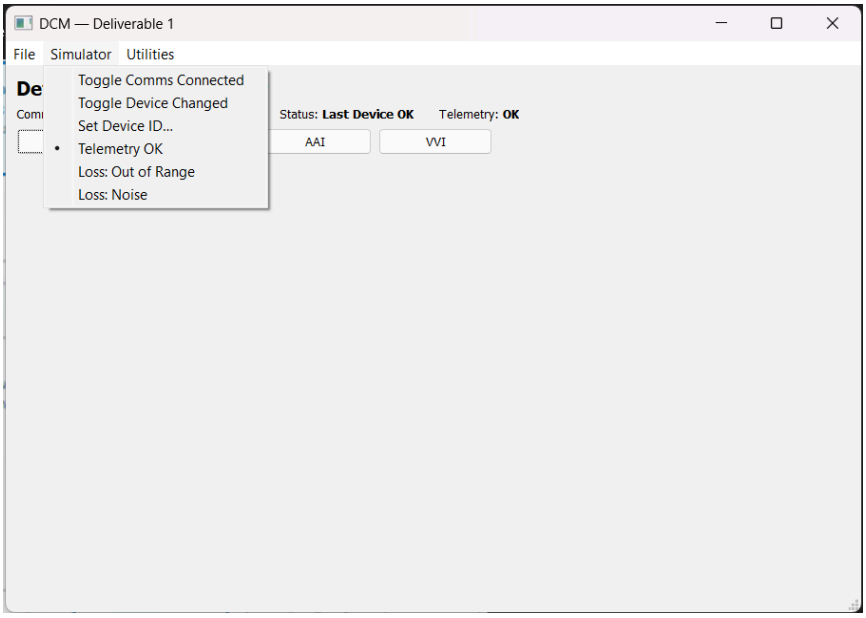

Figure 17: DCM DashBoardPage showcasing the Simulator menu dropdown

Similarly to File, Simulator is a dropdown designed to provide access to various simulation and telemetry testing options. The left side of the dropdown menu utilizes a checkmark to represent “On” and dot to represent the current selection for different simulated components, labeled for user comprehension. The design supports efficient workflow and quick scenario testing in an intuitive manner, accessible throughout the entire interface.

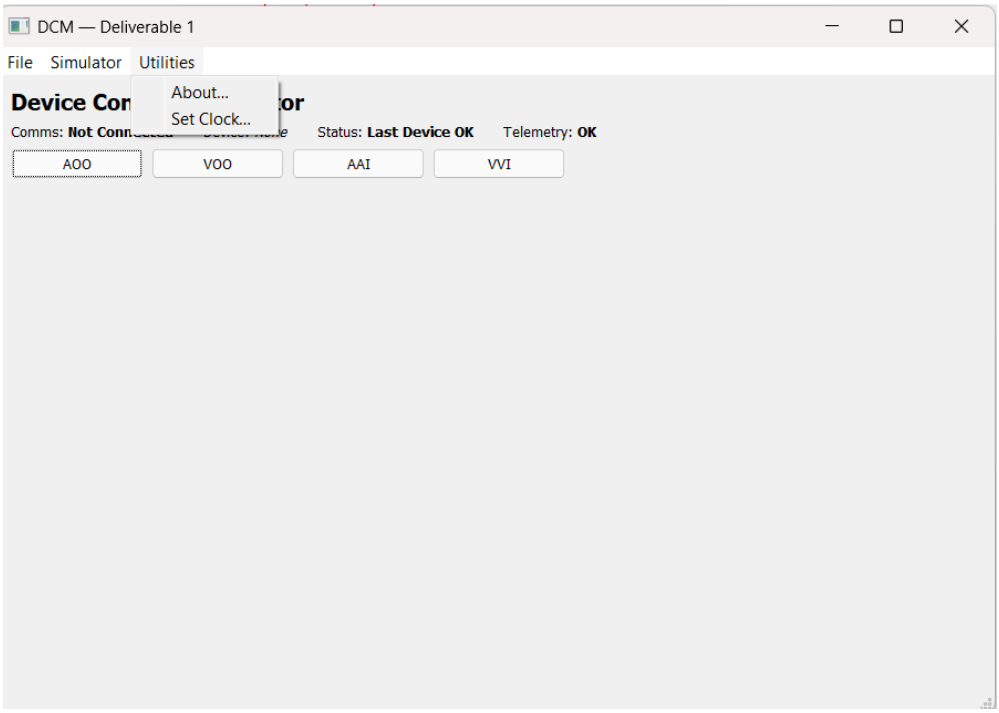

Figure 18: DCM DashBoardPage showcasing the Utilities menu dropdown

The utilities dropdown menu similarly to the File and Simulator is designed to be easily and quickly accessed within all interfaces to either set the clock or display the application information.

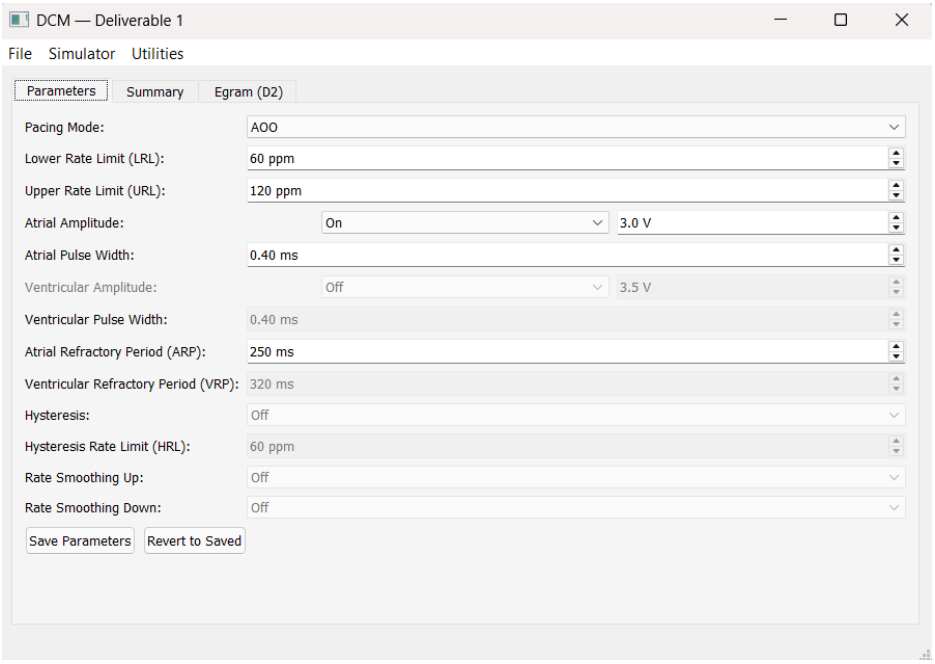

Figure 19: DCM ModeEditorPage Parameter tab for pacing mode AOO

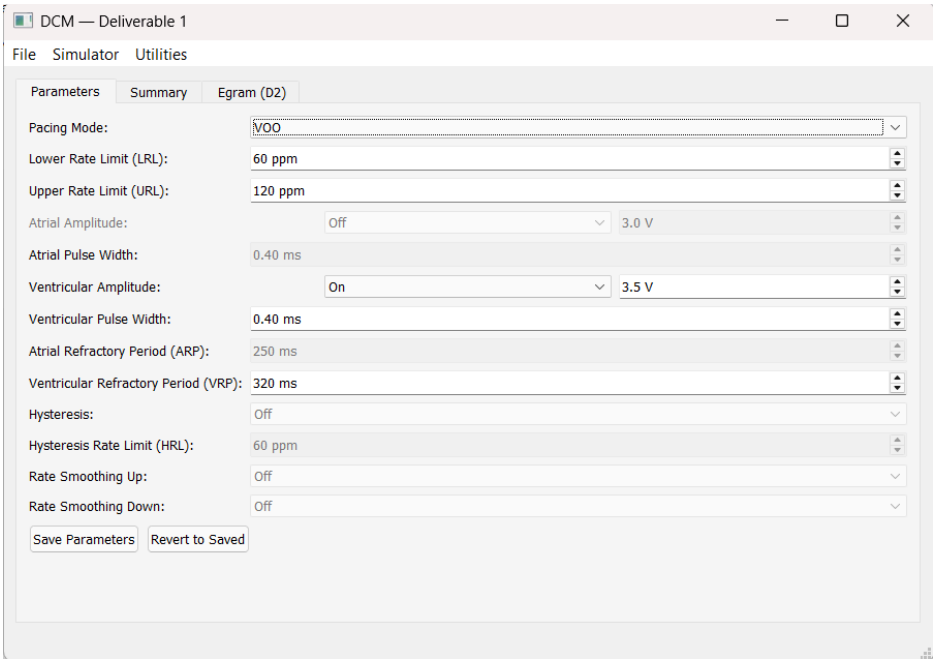

Figure 20: DCM ModeEditorPage Parameter tab for pacing mode VOO

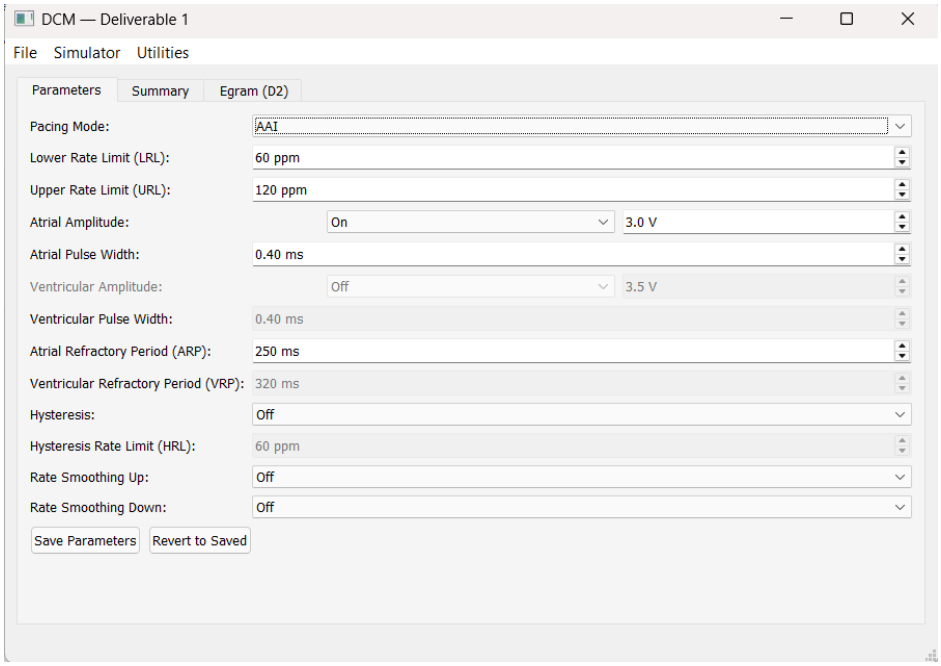

Figure 21: DCM ModeEditorPage Parameter tab for pacing mode AAI

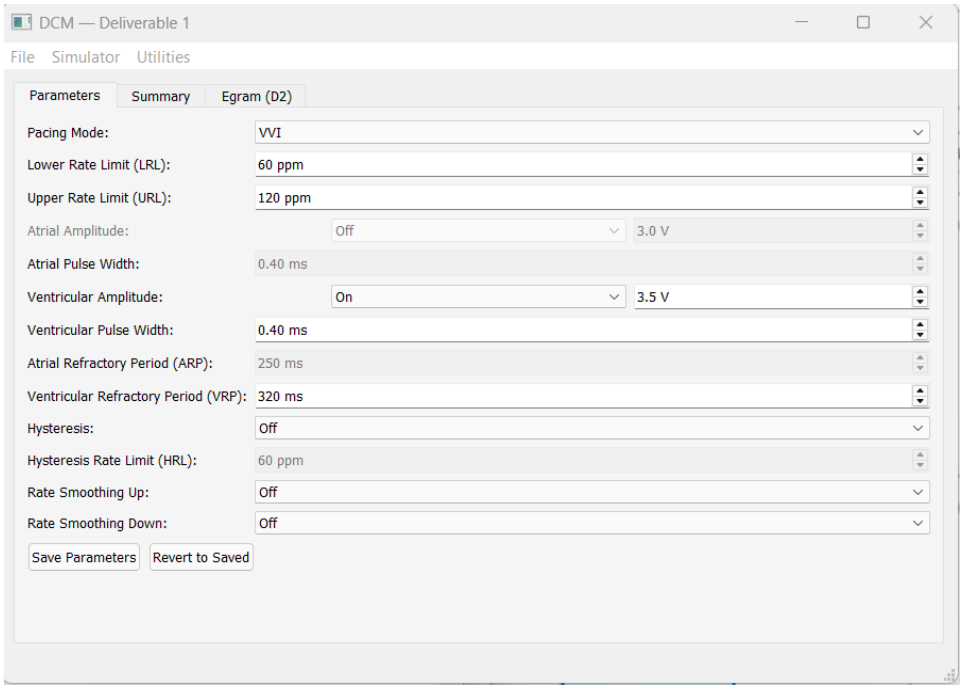

Figure 22: DCM ModeEditorPage Parameter tab for pacing mode VVI

The AOO, VOO, AAI, and VVI sections for within the ModeEditorPage within the parameters tab is designed to utilize dropdown menus due to its intuitive nature and utilizes the proper widgets to ensure only relevant components are enabled and editable by the users. This decreases the likelihood of conflicting data entries with each dropdown only following the accurate step increments for each

parameter. This design is both intuitive and is used to validate for the allowable values entered by the user.

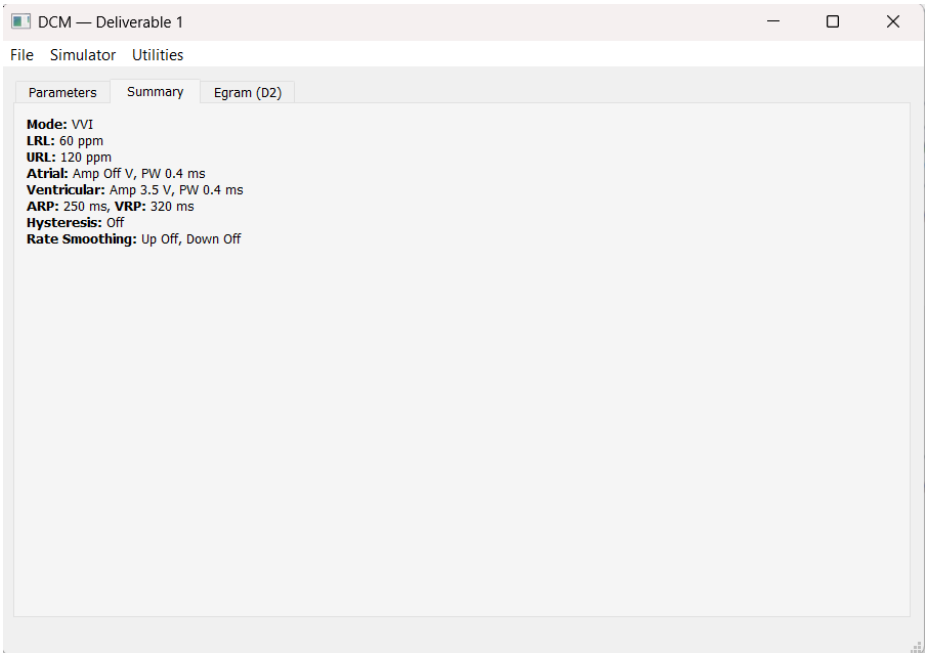

Figure 23: DCM ModeEditorPage Summary tab for VVI with parameters and values

The Summary tab is displayed next to the parameters and Egram tab, designed as an intuitive workflow between the parameters, to the summary, to Egram (visualization of the mode). Furthermore, the parameters are bolded contrasting the values with distinct separation using ":" and "," or line separation. This design enhances user experience and readability for quick interpretation of data.

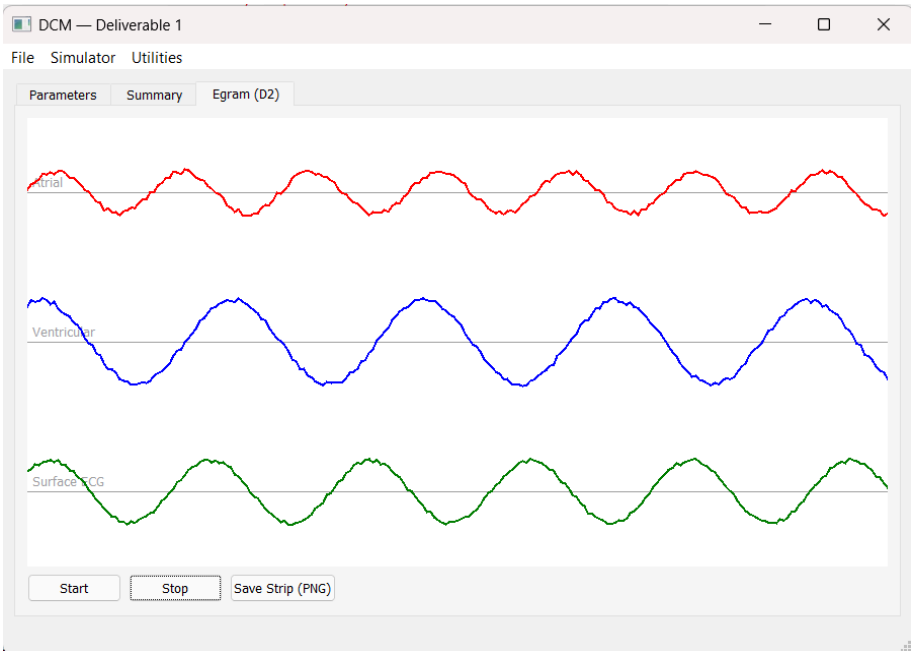

Figure 24: DCM ModeEditorPage Egram tab with atrial, ventricular, and surfaceECG signal visualization

The Egram tab uses clear separation between Atrial, Ventricular, and SurfaceECG waves using different labeled axes and colours. The interactive labeled buttons are intuitive and useful in pausing the signal to enhance readability of the signal, or for starting it again to record more data. The strips can also be saved for the user to review the data in the future. This design aligns with medical software principles and provides an intuitive visualization, enhancing readability.

3. Part 2

3.1. Requirement Potential Changes

3.1.1. Pacemaker Design

Currently, only AOO, VOO, AAI, and VVI modes have been added. In the future, more modes are necessary as stated by the requirements: XOOR, XXIR, and DDDR. Subsequently, interactions between each mode may become more complicated, and several new hardware inputs and outputs will be required for full functionality. Each new module will also be fully integrated into one model with the finished modes of this deliverable.

Rate adaptation is also another parameter that must be implemented for Heartware to be closer to having the full nuance of real-world pacemakers. The pulse rate (LRL) must be able to increase or decrease dynamically in response to detection by the on-board accelerometer. This will provide more flexibility to the implementation of the pacemaker, as this allows the device to adapt to various intensities of exercise.

Hysteresis pacing is another pacing algorithm that may be implemented. This causes the pacemaker to further delay pacing after a sensed beat (increase the wait period) before delivering a pulse, which encourages self-pacing.

3.1.2. Assurance

Given the potential importance of a fully functional pacemaker system, it is absolutely critical to provide assurance that the system is safe. Since a finished version of Heartware will be a safety-critical embedded system, safety assurance cases will be required.

3.1.3. DCM

In the future, additional modes will be added. The DCM will be required to accommodate for these new modes, their parameters and unique features. More buttons and options will be added, making the current DCM code more expansive. Additional, as of this deliverable, the DCM is not required to be connected to the pacemaker. In the future, the connection will likely be required, and the changes and actions within the DCM should be reflected in the pacemaker demonstrations. Additionally, data from the DCM must be converted to a format that can be read by the pacemaker and vice versa. The data the DCM sends to the pacemaker must be checked such that it is within the safe parameters. This should occur both within the DCM and also before the pacemaker software uses the values.

3.2. Design Decision Potential Changes

3.2.1. AOO and VOO

The AOO and VOO modes potentially need to consider other variables from modes implemented in the future, ensuring they are set to false when the mode is selected. Additionally, if magnet-based mode switching is implemented in the future, the modes' interfaces may be altered such that it can interact with other pacing modes/modules

3.2.2. VVI and AAI

The AAI and VVI modes may require modification as additional sensing features such as rate smoothing and hysteresis pacing are implemented. These features will require interval adjustments and additional conditional logic within the Stateflow chart. Furthermore, the XXI mode interfaces may need to be modified if further modes are implemented and to integrate with other modes (existing and new). This includes magnet-based mode switching for transitioning between XXI/XOO modes.

3.2.3. DCM

With the addition of integration with the physical pacemaker hardware, several design decisions may need to be reevaluated to accommodate the new functionality and communication requirements. For instance, the current design uses a JSON file for data collection. However, the integration may require real-time communication, and thus real-time updating data. The JSON file can be kept for saved data, but a new type of data storing might need to be added to account for telemetry and connection as well. This will likely be done by modifying the Database and MainWindow classes to separate file-based storage. The data must also be called and used for the Egram (D2) tab to display real graphs, rather than ones based on simulated values.

Additionally, with the addition of new modes, this means new widgets and buttons that lead to new pages, dealt with by ModeEditorPage. The same logic for the pages can likely be used and modified accordingly.

The interface must also be modified to add error messages related to hardware and connections. Additionally, if new parameters are added or changed, the validation functions and parameter widgets will need to be modified accordingly.

Overall, the PyQt5-based structure should still be sufficient for this project, but if need be, the information and structure for the DCM can be taken and adapted for other platforms or languages based on need,

3.3. Module Description

The pacemaker modes do not have any implemented interactions with other components, however, they do have the ability to switch between states for demonstration purposes.

3.3.1. AOO

The AOO module provides consistent, periodic pacing to the atrium without sensing signals from the heart.
It periodically discharges a capacitor to pace the atrium, then recharges the capacitor for the next pacing cycle.
Global variables: p_aPaceAmp, p_aPaceWidth (input variables, shared amongst the AXX modes)
State variables: max_V (maximum nominal voltage the capacitor can discharge)

3.3.2. VOO

The VOO module is functionally identical to the AOO, with the difference being that it supports sensorless, periodic pacing of the ventricle instead of the atrium.
This process occurs through the discharging of a blocking capacitor to pace the ventricle, then recharging another capacitor for the next pacing period.
Global variables: p_vPaceAmp, p_vPaceWidth (input variables shared amongst the VXX modes)
State variables: max_V

3.3.3. VVI

The VVI module provides pacing to the ventricle with intrinsic beat sensing and pulse delivery inhibition. It monitors natural ventricular activity and only delivers a pulse when no intrinsic ventricular beat is sensed within the programmed interval. It cycles between charging and discharging of capacitors to deliver current through the ventricle and ensure a net zero current through the heart.
Global variables: p_vPaceAmp, p_vPaceWidth (input variables shared amongst the VXX modes)
State variables: max_V

3.3.4. AAI

The AAI module provides pacing to the atrial with intrinsic beat sensing and pulse delivery inhibition. It monitors natural atrial activity and only delivers a pulse when no intrinsic atrial beat is sensed within the programmed interval. It cycles between charging and discharging of capacitors to deliver current through the atrium and ensure a net zero current through the heart.
Global variables: p_aPaceAmp, p_aPaceWidth (input variables shared amongst the AXX modes)
State variables: max_V

3.3.5. DCM

The DCM uses several classes, methods, and global variables to create a reliable, interactive interface.

3.3.5.1. Global Variables & Constants

There are several global variables and constants used throughout the DCM code. They act as a single source for unchangeable, hidden values and information.

DB_FILE = “dcm_d1_file.json”: this is the path to the JSON file with all the usernames, passwords and saved parameters. In the code, the file is called “dcm_d1_file.json.” The variable is used by the Database class within the MainWindow class to point at the JSON file.

MAX_USERS = 10: As per the DCM requirements, the pacemaker interface cannot account for more than 10 unique logins - username and password combinations. The value of maximum users is set as global to ensure it is not modified, and can be easily referenced. This variable is used in the Database class, within the add_user function - if the number of users is equal to or larger than MAX_USERS, no more user information can be added. It is also used within the _handle_register function in the LoginPage class, sending an error message if the maximum 10 users is reached.

APP_MODEL_NUMBER = "DCM D1": This constant is used for the AboutDialog class to form the About popup window. For Deliverable 1, this is set to any random value,

APP_SOFTWARE_REV = "D1": This constant is used for the AboutDialog class to form the About popup window. For Deliverable 1, this is set to any random value,

APP_SERIAL_NUMBER = "SN": This constant is used for the AboutDialog class to form the About popup window. For Deliverable 1, this is set to any random value,

APP_INSTITUTION = "McMaster University": This constant is used for the AboutDialog class to form the About popup window. For Deliverable 1, this is set to McMaster University,

3.3.5.2. Programmable Parameters:

As per Appendix A in System Specifications in the Pacemaker (D1) document, there are certain programmable parameters with set ranges and step values.

LRL_SEGMENTS = [(30, 50, 5), (50, 90, 1), (90, 175, 5)] : this is the piecewise specification for the lower rate limit variable, in ppm. This variable is used by the helper function, build_allowed_lrl, to provide an acceptable range of values to be worked with on the interface.

URL_MIN, URL_MAX, URL_STEP = 50, 175, 5 : this is the piecewise specification for the lower rate limit variable, in ppm. These variables are used by the helper function, build_allowed_url, to provide an acceptable range of values to be worked with on the interface.

AMP_LOW_MIN, AMP_LOW_MAX, AMP_LOW_STEP = 0.5, 3.2, 0.1: These variables define the minimum, maximum, and step for the amplitude (in Volts) for the Atrium. They are used by the AmplitudeWidget class within the ModeEditorPage class to set the acceptable values on the page. This is used to create another global variable, ALLOWED_LRL.

AMP_HIGH_MIN, AMP_HIGH_MAX, AMP_HIGH_STEP = 3.5, 7.0, 0.5: These variables define the minimum, maximum, and step for the amplitude (in Volts) for the Ventricle. They are used by the AmplitudeWidget class within the ModeEditorPage class to set the acceptable values on the page. This is used to create another global variable, ALLOWED_URL.

PW_SEGMENTS = [(0.05, 0.1, 0.05), (0.1, 1.9, 0.1)]: this is the piecewise specification for the pulse width variable, in ppm. These variables are used by the helper function, build_pw_values, to provide an acceptable range of values to be worked with on the interface. They are used to create another global variable, PW_VALUES.

REF_MIN, REF_MAX, REF_STEP = 150, 500, 10: These values are used to create another global variable REF_VALUES, stating the allowed, inclusive range and step for the refractory period.

MODES = ["AOO", "VOO", "AAI", "VVI"]: These are the modes used in D1, in a list of strings. The MODES variable is used in the DashboardPage class to create buttons to choose the desired mode to program in. It is also used in the ModeEditorPage class to set a default mode as the value in the first index - AOO.

HYSTERESIS_STATES = ["Off", "On"]: Hysteresis can be set to “Off” or “On.” If it is on, the values of hysteresis are the same as those in LRL. This variable is used in the ModeEditorPage class to add them as options to change during the programming.

RATE_SMOOTH_CHOICES = ["Off", "3%", "6%", "9%", "12%", "15%", "18%", "21%", "25%"]: The rate smoothing can be “Off” or “On.” If it is on it can be up or down. These options use this global variable within the ModeEditorPage class.

3.3.5.3. Interaction Summary

All custom widgets in the DCM read the values in the global variables to constrain input and changes. ModeEditorPage uses them to enable and disable fields per mode and build defaults, with the help of AmplitudeWidget’s allowed value creation. MainWindow creates files and reports, and along with LoginPage manages the JSON file. DashboardPage creates appropriate buttons. Finally, AboutDialog creates the About popup tab using set information.

3.3.5.4. Classes & Methods

3.3.5.4.1. Database:

The Database class encapsulates storage for users and user-specific parameter sets. Using a *Dict* format, users can be stored in the JSON file along with their passwords and parameters per mode. This is the only component that knows about and interferes with the file system, keeping the file itself hidden from the interface. It is constructed with a public __init__(path: str) that stores self.path, pointing to DB_FILE in the file location, and if the file does not exist, writes a skeleton {“users”: [], “params”:{}}. It has private _read() and _write(data) functions that are the only JSON entry and modification paths in the code, helping hide it.

It also features public functions such as user_count() that returns an integer for capacity checks and add_user(username, password_hash) → bool, that enforces case-insensitive uniqueness and the MAX_USERS capacity. Additionally, verify_user(username, password_hash) → bool is another public method that confirms the username and password match. These are the only functions that the class LoginPage will need to implement login and registration.

The public save_params(username, mode, params) and load_params(username, mode) can write and retrieve a parameter dictionary as a key “<user>:<mode>.” These are called by ModeEditorPage to enable their functions in different pages and modes. Here, the public methods were assigned as such because other modules must call for them; read/write methods are private to keep changes within the class.

3.3.5.4.2. Custom Editors (LRLSpinBox, URLSpinBox, PWSpinBox,RefPeriodSpinBox)

The classes are created, taking form QSpinBox, to create drop-down menus for the input and modification of parameter values. They are public classes and are used by the ModeEditorPage class that constructs them directly and creates their features on the mode page. The key functions in each are __init__ for initialization, stepBy for increasing and decreasing the parameter value using the up and down arrows, in accordance with the step counts within the allowed range, and additionally snapping values to their closest allowed value. The validate function controls what is accepted when users type inside the box, using QValidator’s intermediate, invalid, and acceptable states. RefPeriodSpinBox works a little differently and doesn't need to override stepping or validation as it has a continuous range. These classes are instantiated and owned by ModeEditorPage and their domain enforcement is used for _handle_save and the Summary and reports. The methods are public because Qt expects to call them through the event system. Additionally, the data they rely on (self.allowed) are based on the allowed lists created from the global variables set for the programmable variables.

3.3.5.4.3. AmplitudeWidget:

The AmplitudeWidget class is a public composite control that encapsulates the Off/On semantics and voltage entry for either the atrial or ventricular modes into a single row that will return “Off” or a value snapped and clamped to the allowed range. It has durable state variables (_min_v, _max_v, _step_v) that define the chamber-specific domain. These are internal changes that should not be modified by outside code after construction. It has public children, label, state_combobox, volt_spinbox that can be referenced elsewhere. It has a public value() function that returns “Off” or a snapped float. setValue(val) accepts “Off” or a float and configures both controls, and _update_enabled_state() is a private function connected to currentIndexChanged within the __init__that keeps the voltage field enabled when “On” is selected. This class is instantiated twice by the ModeEditorPage class (atrial and ventricular) with chamber-specific domains. It is also used by _collect_params and _apply_params_to_widgets and defaults logic.

3.3.5.4.4. LoginPage:

LoginPage is a class that works as the authentication at the start of the system to get to the rest of the features, providing a clean, minimal UI for user registration, password and hashed password creation, and login. On construction, it uses QLineEdit to allow for user input. Its key private methods are _handle_register(), which validates presence, capacity and uniqueness, then sets a hashed password, and _handle_login(), which hashes the attempted password and calls db.verify_user. On success, it calls the callback passed from MainWindow. The class holds references to db and the on_login_ok callback. This is a public class constructed and embedded by MainWindow. It calls MainWindow._on_login_ok(username) to transition to the stacked widget, depending on the hash_password and database for credential logic.

3.3.5.4.5. DashboardPage:

DashboardPage is a public class that can be created by MainWindow. It provides the hub screen after logging in, mirrors the simulator state (comms/device/change/telemetry) and takes the user to a chosen pacing mode. It gives a good overview of what's available. Its key public functions are show_comms(bool), show_device(str), show_changed(bool), and show_telemetry(state:str) to show the four status labels. The Button clicked signals call on_mode_click(mode) to choose, from the four mode buttons created using QLabel. It also receives updates from MainWindow whenever the simulator changes state and invokes MainWindow._open_mode_editor(mode) when a mode button is pressed.

3.3.5.4.6. D1EgramView:

D1EgramView is a paint-driven viewer that simulates atrial, ventricular and surface ECG motions to demonstrate egram capabilities. Its key public functions are start() and stop(), starting a timer and stropping the motions. The internal _tick() function advances time and appends noisy samples to self.atrialList, self.ventricularList, and self.surfaceList, trims each list to self.buffer_len, and calls update() to schedule a repaint. The paintEvent function draws baselines and labels for visible traces, it reads from state variables like buffer_len, dt, and the three public sample lists. _tick itself is private as no other module should be advancing the simulation manually.

3.3.5.4.7. AboutDialog:
The AboutDialog class uses QDialog to create a popup window to display required information pertaining to the project, from the global variables APP_MODEL_NUMBER, APP_SOFTWARE_REV, APP_SERIAL_NUMHER, and APP_INSTITUTION. These are all made using QLabel. Only the constructor initializes a simple form with an OK button. It is launched from MainWindow._show_about() and is thus public.

3.3.5.4.8. SetClockDialog:
The SetClockDialog allows the user to set the DCM "device clock" displayed in the status bar. It is a popup window relying on QDialog, in the utilities section with AboutDialog. The constructor seeds a QDateTimeEdit with the current device time and configures the OK/Cancel. It also uses the public function of selected_datetime() to expose the chosen time. This class is launched from MainWindow._set_clock_dialog(). On acceptance, MainWindow also updates device_clock and calls _refresh_status_bar().

3.3.5.4.9. ModeEditorPage:
ModeEditorPage is a public class that owns all parameter editors, the Summary tab, and the Egram tab, and interacts with the database and MainWindow. Its public set_mode(mode: str) is the main entry point, setting self.current_mode, synchronizing the visible QComboBox and enabling and disabling the appropriate widgets according to the mode. It also calls the private _handle_revert() to load saved values for the active user or apply defaults. _handle_save () is another internal implementation. It verifies an active user via the public callback get_active_user(), rejects LRL > URL, gathers parameters with _collect_params(), continues with db.save_params and refreshes Summary. _handle_revert asks db.load_params(user, mode) and either applies them with _apply_params_to_widgets(p) or calls _apply_defaults_for_mode(mode) and rebuilds the Summary. The private _collect_params() returns a dict with normalized fields and uses _percent_to_int to return the smoothing as integers. Additionally,

_apply_params_to_widgets(p) is a private method that writes values into each widget, essentially doing the reverse.

_refresh_summary() rebuilds a read-only HTML view of the current parameters. It is private and cannot be called by external callers to manipulate Summary directly. Other key features are _on_tab_changed(idx), a private slot tied to QTabWidget.currentChanged: it starts the egram when the tab becomes visible and stops it otherwise. Additionally, _save_egram_png() is a private detail that saves a screenshot of the D1EgramView. The principal state variables are current_mode, the widget instances for each parameter, and the egram view. They are public instance attributes inside the class to simplify testing and to allow Qt signal connections, but the behaviours that mutate them are exposed through the public set_mode and button clicks to keep a clear surface. No globals are changed.

MainWindow gets the active user, and then this class reads and writes parameters via Database, controls D1EgramView and provides current parameter snapshots to MainWindow report generators.

3.3.5.4.10. EgramData:

The EgramData class is an Egram page placeholder to eventually stream data in D2. The constructor stores time and the atrial and ventricular channels., and __repr__ provides useful debug logging. There is no public behavior yet.

3.3.5.4.11. MainWindow:
The MainWindow class is the public class that essentially orchestrates the entire application. It shows the public callbacks used by children, such as _on_login_ok(username) (called by LoginPage), _open_mode_editor(mode) (called by DashboardPage), and _get_active_user() (called by ModeEditorPage), all of which are private to the application shell. The class also provides a public menu with slot methods for simulator toggles (_toggle_comms, _toggle_device_changed, _set_device_id, _set_telemetry) and utilities (_show_about, _set_clock_dialog), which are UI handlers. There are also private reporting/export helpers _current_params(), _report_header_html(report_name), _params_table_html(p), _save_pdf(html, suggested), _export_brady_params_report(), _export_temp_params_report(), and _export_all_json() that orchestrate UI actions and file dialogs, taking plain values and returning nothing but accomplishing their intended goals of creating reports or saving files.

The state variables on MainWindow are public instance attributes: db, active_user, comms_connected, device_id, device_changed, telemetry_state, device_clock, stack, status_bar. Additionally, here,. _refresh_status_bar() is internal and re-composes the footer line from the current state. It is private to allow for change in formatting without breaking collars, while exposing similar actions.

3.3.5.4.12. Entry Point - if __name__
This is public module behaviour that creates the actual application using QApplication(), constructs MainWindow and calls show() and enters the event loop with app.exec(). There are no state globals here because MainWindow owns everything after construction. This is what runs the program.

3.4. Testing

3.4.1 AOO and VOO Modes

The XOO modes were tested by varying the input parameters and observing the output to the heart on the Heartview software. In order for the result of a test to be ‘pass’, all outputs must be within the tolerance range of the expected value. For p_xooInterval, this is within 8mS of the input value, for p_xPaceWidth, this is within 0.2mS of the input value, and for p_xPaceAmp, this is within 12% of the input value. All three output parameters must lie within these ranges for the mode to pass the test, as it is critical the correct pacing is delivered.

Images of the Heartview output for each test are included in the appendix.

Purpose: ensure consistent and correct pacing by default

p_aooInterval	**p_aPaceWidth**	**p_aPaceAmp**	**Result**
1000mS (default)	0.4mS (default)	3500mV (default)	Pass

p_vooInterval	**p_vPaceWidth**	**p_vPaceAmp**	**Result**
1000mS (default)	0.4mS (default)	3500mV (default)	Pass

Purpose: check response to changes in desired interval

p_aooInterval	**p_aPaceWidth**	**p_aPaceAmp**	**Result**
2000mS	0.4mS (default)	3500mV (default)	Pass
500mS	0.4mS (default)	3500mV (default)	Pass

p_vooInterval	**p_vPaceWidth**	**p_vPaceAmp**	**Result**
2000mS	0.4mS (default)	3500mV (default)	Pass
500mS	0.4mS (default)	3500mV (default)	Pass

Purpose: check response to changes in desired pace width

p_aooInterval	**p_aPaceWidth**	**p_aPaceAmp**	**Result**
1000mS (default)	0.2mS	3500mV (default)	Pass
1000mS (default)	0.8mS	3500mV (default)	Pass

p_vooInterval	p_vPaceWidth	p_vPaceAmp	Result
1000mS (default)	0.2mS	3500mV (default)	Pass
1000mS (default)	0.8mS	3500mV (default)	Pass

Purpose: check response to changes in desired pace amplitude

p_aooInterval	p_aPaceWidth	p_aPaceAmp	Result
1000mS (default)	0.4mS (default)	5000mV	Pass
1000mS (default)	0.4mS (default)	2000mV	Pass

p_vooInterval	p_vPaceWidth	p_vPaceAmp	Result
1000mS (default)	0.4mS (default)	5000mV	Pass
1000mS (default)	0.4mS (default)	2000mV	Pass

3.4.2 AAI and VVI Modes

The XXI modes were tested by varying the input parameters and observing the output to the heart on the Heartview software. In order for the result of a test to be ‘pass’, all outputs must be within the tolerance range of the expected value. For p_lowrateInterval, this is within 8mS of the input value, for p_xPaceWidth, this is within 0.2mS of the input value, and for p_xPaceAmp, this is within 12% of the input value. All three output parameters must lie within these ranges for the mode to pass the test, as it is critical the correct pacing is delivered. As well, the functionality of the pacemaker was tested by varying the input heart rate at the default settings to observe if the circuit correctly detects the rising edge of the pulses. As well, constant values, 0 and 1 were inputted, to observe how the circuit behaves when it either never senses a pulse or always does, respectively. The expected results require the pacemaker output to lag any detected natural heart pulses by at least the refractory period.

Images of the Heartview output for each test are included in the appendix.

Purpose: ensure consistent and correct pacing by default

p_lowrateInterval	p_aPaceWidth	p_aPaceAmp	Result
1000mS (default)	0.4mS (default)	3500mV (default)	Pass

p_lowrateInterval	p_vPaceWidth	p_vPaceAmp	Result
1000mS (default)	0.4mS (default)	3500mV (default)	Pass

Purpose: check response to changes in desired interval

p_lowrateInterval	p_aPaceWidth	p_aPaceAmp	Result
2000mS	0.4mS (default)	3500mV (default)	Pass
500mS	0.4mS (default)	3500mV (default)	Pass

p_lowrateInterval	p_vPaceWidth	p_vPaceAmp	Result
2000mS	0.4mS (default)	3500mV (default)	Pass
500mS	0.4mS (default)	3500mV (default)	Pass

Purpose: check response to changes in desired pace width

p_lowrateInterval	p_aPaceWidth	p_aPaceAmp	Result
1000mS (default)	0.2mS	3500mV (default)	Pass
1000mS (default)	0.8mS	3500mV (default)	Pass

p_lowrateInterval	p_vPaceWidth	p_vPaceAmp	Result
1000mS (default)	0.2mS	3500mV (default)	Pass
1000mS (default)	0.8mS	3500mV (default)	Pass

Purpose: check response to changes in desired pace amplitude

p_lowrateInterval	p_aPaceWidth	p_aPaceAmp	Result
1000mS (default)	0.4mS (default)	5000mV	Pass
1000mS (default)	0.4mS (default)	2000mV	Pass

p_lowrateInterval	p_vPaceWidth	p_vPaceAmp	Result
1000mS (default)	0.4mS (default)	5000mV	Pass
1000mS (default)	0.4mS (default)	2000mV	Pass

Purpose: check if the pacemaker behaves correctly based on whether a heart beat is sensed

p_lowrateInterval	p_aPaceWidth	p_aPaceAmp	m_as	Result
1000mS (default)	0.4mS (default)	3500mV (default)	0 constant (no	Pass

			signal detected)	
1000mS (default)	0.4mS (default)	3500mV (default)	1 constant (signal always detected)	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	Input Heart Rate (HR) of 60bpm	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	Input HR of 30bpm	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	Input HR of 120 bpm	Pass

p_lowrateInterval	**p_vPaceWidth**	**p_vPaceAmp**	**m_vs**	**Result**
1000mS (default)	0.4mS (default)	3500mV (default)	0 constant (no signal detected)	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	1 constant (signal always detected)	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	Input Heart Rate (HR) of 60bpm	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	Input HR of 30bpm	Pass
1000mS (default)	0.4mS (default)	3500mV (default)	Input HR of 120 bpm	Pass

3.4.3. DCM

Purpose: check new user registration max capacity, duplicates, and if valid inputs

Output legends:

A. “Please enter a name and password” dialog
B. “User exists or capacity reached.” dialog
C. “Registration complete please login” dialog
D. “Maximum of 10 users stored.” dialog

Name	Password	Expected Output	Actual Output	Pass/Fail
(leave blank)	(leave blank)	A	A	Pass
Test1	(leave blank)	A	A	Pass
(leave blank)	Password1	A	A	Pass

Test1	Password1	C	C	Pass
Test1	Password1	B	B	Pass
Test1	(leave blank)	A	A	Fail
Test2	Password2	C	C	Pass
Test3	Password3	C	C	Pass
Test4	Password4	C	C	Pass
Test5	Password5	C	C	Pass
Test6	Password6	C	C	Pass
Test7	Password7	C	C	Pass
Test8	Password8	C	C	Pass
Test9	Password9	C	C	Pass
Test10	Password10	C	C	Pass
Test11	Password11	D	D	Pass

Purpose: Check login if it matches login credentials with either valid or invalid passwords for non-existent or existing users.

Output legends:

A. “Invalid username or password” dialog.
B. Changes from the LoginPage to the DashboardPage with “Logged in as <Name> shown at the bottom left of the window.

Name	Password	Expected Output	Actual Output	Pass/Fail
(leave blank)	(leave blank)	A	A	Pass
Test1	(leave blank)	A	A	Pass
(leave blank)	Password1	A	A	Pass
Test1	FalsePass1	A	A	Pass
FakeTest1	(leave blank)	A	A	Pass
FakeTest2	FakePass2	A	A	Pass
Test1	Password1	B	B	Pass

Test2	Password2	B	B	Pass
Test3	Password3	B	B	Pass
Test4	Password4	B	B	Pass
Test5	Password5	B	B	Pass
Test6	Password6	B	B	Pass
Test7	Password7	B	B	Pass
Test8	Password8	B	B	Pass
Test9	Password9	B	B	Pass
Test10	Password10	B	B	Pass
Test11	Password11	A	A	Pass
Password1	Test1	A	A	Pass

Purpose: test File dropdown menu functions, security, and data and report export.
Output Legend:

A. “Please log in and open the Mode Editor first” dialog
B. Opens files and allows for the JSON file “dcm_params” to be downloaded
C. Opens files and allows for the .PDF file “Bradycardia Parameters Report” to be downloaded with a dialog “PDF saved to: C:/Path/to/file/Bradycardia Parameters Report.pdf after “Save is confirmed within the file.
D. Opens files and allows for the .PDF file “Temporary Parameters Report” to be downloaded with a dialog “PDF saved to: C:/Path/to/file/Temporary Parameters Report.pdf after “Save is confirmed within the file.

File dropdown	On the Page	Expected dialog/output	Actual dialog/output	Pass/Fail
Reports-> Bradycardia parameter reports	LoginPage	A	A	Pass
Reports-> Temporary parameter reports	LoginPage	A	A	Pass
Export Saved Params JSON	LoginPage	B	B	Pass
Reports->	DashboardPage	C	C	Pass

Bradycardia parameter reports				
Reports-> Temporary parameter reports	DashboardPage	D	D	Pass
Export Saved Params JSON	DashboardPage	B	B	Pass
Reports-> Bradycardia parameter reports	ModeEditorPage	C	C	Pass
Reports-> Temporary parameter reports	ModeEditorPage	D	D	Pass
Export Saved Params JSON	ModeEditorPage	B	B	Pass

Purpose: test Simulator dropdown menu for DashboardPage device controller monitor for device communication and telemetry.

Output Legend:

A. Status at bottom bar reads: “Connected” before the “| Device:” section.
B. Status at bottom bar reads: “Not Connected” before the “| Device:” section.
C. Status at bottom bar reads: “Last Device OK” after the “| Device:” section.
D. Status at bottom bar reads: “Device Changed” after the “| Device:” section.
E. Status at bottom bar reads: “| Device: None”
F. Open a file to Enter device ID then Cancel or Save and the File name can be changed and will display whatever file name is entered by the user in the status at the bottom bar reading: “ | Device: <file name>” section.
G. Status at bottom bar reads: “Telemetry: OK”
H. Status at bottom bar reads: “Telemetry: Lost-Out of Range
I. Status at bottom bar reads: Telemetry: Lost-Noise
J. On DashboardPage underneath Device Controller-Monitor: “Comms: Not Connected”
K. On DashboardPage underneath Device Controller-Monitor: “Comms: Connected”
L. On DashboardPage underneath Device Controller-Monitor: “Status: Last Device OK”
M. On DashboardPage underneath Device Controller-Monitor: “Status: Device Changed”
N. On DashboardPage underneath Device Controller-Monitor: “Device: *None*”
O. Open a file to Enter device ID then Cancel or Save and the File name can be changed and will display whatever file name is entered by the user displayed on the DashboardPage underneath Device Controller-Monitor: “Device: <file name>”
P. On DashboardPage underneath Device Controller-Monitor: “Telemetry: OK”
Q. On DashboardPage underneath Device Controller-Monitor: “Telemetry: Lost - Out of Range”
R. On DashboardPage underneath Device Controller-Monitor: “Telemetry: Lost - Noise”

Simulator Toggled (clicked on)	On the Page	Expected dialog/output	Actual dialog/output	Pass/Fail
Toggle Comms Connected (“Checked On”)	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	A	A	Pass
Toggle Comms Connected (“Checked Off”)	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	B	B	Pass
Toggle Device Changed (“Checked Off”)	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	C	C	Pass
Toggle Device Changed (“Checked Off”)	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	D	D	Pass
Set Device ID… (default state with no user filename entry)	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	E	E	Pass
Set Device ID… (user filename entry)	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	F	F	Pass
Telemetry OK	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	G	G	Pass
Loss: Out of Range	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	H	H	Pass
Loss: Noise	At the bottom left of the LoginPage, DashBoardPage, and ModeEditorPage	I	I	Pass
Toggle Comms Connected (“Off”)	On the Dashboard page underneath DCM	J	J	Pass

Toggle Comms Connected (“On”)	On the Dashboard page underneath DCM	K	K	Pass
Toggle Device Changed (“Off”)	On the Dashboard page underneath DCM	L	L	Pass
Toggle Device Changed (“On”)	On the Dashboard page underneath DCM	M	M	Pass
Set Device ID… (default state with no user filename entry)	On the Dashboard page underneath DCM	N	N	Pass
Set Device ID… (user filename entry)	On the Dashboard page underneath DCM	O	O	Pass
Telemetry OK	On the Dashboard page underneath DCM	P	P	Pass
Loss: Out of Range	On the Dashboard page underneath DCM	Q	Q	Pass
Loss: Noise	On the Dashboard page underneath DCM	R	R	Pass

Note: Only the LoginPage will be displayed for the outputs within the appendix as all 3 pages will have the same output in the same location.

Purpose: test Utilities dropdown menu function and information.

Output Legend:

A. An About DCM dialog appears with “Application model number: DCM D1, Software revision: D1, DCM serial number: SN, and Institution name: McMaster University”
B. The bottom left on each page should display the default current time in the format “yyyy-mm-dd hh:mm:ss”
C. A Set Clock dialog appears with “Device date/time:” with the format “yyyy-mm-dd hh:mm:ss” editable by typing it
D. A drop down menu for the selectable calendar days displaying the current/saved date in Set Clock
E. The bottom left on each page should display the newly edited time in the format “yyyy-mm-dd hh:mm:ss”

Utilities (clicked on and action)	On the Page	Expected dialog/output	Actual dialog/output	Pass/Fail
About DCM (clicked)	LoginPage, DashboardPage, and ModeEditorPage	A	A	Pass
(Status bar at the bottom of the window for each	At the bottom left of the LoginPage,	B	B	Pass

page)	DashBoardPage, and ModeEditorPage			
Set Clock (clicked)	LoginPage, DashboardPage, and ModeEditorPage	C	C	Pass
Set Clock -> dropdown next to libraries (clicked)	LoginPage, DashboardPage, and ModeEditorPage	D	D	Pass
(After clock is set, status bar appears at the bottom left of the window)	LoginPage, DashboardPage, and ModeEditorPage	E	E	Pass

Note: It'll be displayed on all the pages so the test encompasses all pages.

Purpose: Pacing mode test on the ModeEditorPage with the different modes, the page, expected and actual output, pass/fail:

Output Legend:

A. Displays AOO pacing mode in the ModeEditor on the Parameters tab
B. Displays VOO pacing mode in the ModeEditor on the Parameters tab
C. Displays AAI pacing mode in the ModeEditor on the Parameters tab
D. Displays VVI pacing mode in the ModeEditor on the Parameters tab
E. Displays the accurate default values (if there are no saved parameters) matching the requirements with LRL as 60ppm, URL as 120 ppm, Atrial and Ventricular Amplitude as 3.0 V and 3.5V, Atrial and Ventricular PW as 0.40 ms, ARP as 250 ms, VRP as 320 ms, and HRL as 60 ppm.
F. Atrial Amplitude is manually turned "Off" only in the AOO and AAI pacing modes, disabling the editability of the numeric values associated with it.
G. Ventricular Amplitude is manually turned "Off" only in the VOO and VVI pacing modes, disabling the editability of the numeric values associated with it.
H. HRL is editable
I. HRL is not editable
J. Rate Smoothing Up dropdown menu following the requirements with the values "Off", "3%", "6%", "9%", "12%", "15%", "18%" "21%" and "25%".
K. Rate Smoothing Down dropdown menu following the requirements with the values "Off", "3%", "6%", "9%", "12%", "15%", "18%" "21%" and "25%".
L. Lowest values possible for all parameters with LRL as 30 ppm, URL as 50 ppm, Atrial amplitude as 0.5V and ventricular amplitude as 3.0V, atrial and ventricular pulse width as 0.05ms, ARP and VRP as 150 ms, and HRL as 30 ppm.
M. Highest values possible for all parameters with LRL as 175 ppm, URL as 175 ppm, Atrial amplitude as 3.2V, ventricular amplitude as 7.0 V, Atrial and ventricular pulse width as 1.90 ms, ARP and VRP as 500 ms, and HRL as 175 ppm.
N. The step increments for all parameters should follow the requirements and design decision parameters.

O. Should be displayed similarly to the test shown in test appendix with Mode, LRL, URL, Atrial, Ventricular, ARP, Hysteresis, and Rate Smoothing matching the saved parameters the user is on.
P. Display an Egram with the Atrial, Ventricular, and Surface ECG signals in separate labeled axis with a Start, Stop, and Save Strip (PNG) button that starts, stops, and saves the signal.
Q. Saved the parameters into the local dcm_d1_file and can be accessed upon opening
R. Reverts the parameters to the previously saved parameters or to default if there are no saved parameters.

Mode/Action	On the Page	Expected Output	Actual Output	Pass/ Fail
Click AOO/VOO/AAI/VVI form DashboardPage	DashboardPage	A/B/C/D, respectively and E	A/B/C/D, respectively and E	Pass
Change pacing modes using dropdown to AOO/VOO/AAI/VVI	ModeEditorPage -> Parameters tab	A/B/C/D, respectively and E	A/B/C/D, respectively and E	Pass
Atrial Amplitude and Atrial Pulse Width fields are enabled and editable.	ModeEditorPage ->Parameters tab	A and C (only for AOO and AAI)	A and C (only for AOO and AAI)	Pass
Ventricular Amplitude and Ventricular Pulse Width fields are enabled and editable.	ModeEditorPage ->Parameters tab	B and D (only for VOO and VVI)	B and D (only for VOO and VVI)	Pass
Atrial or Ventricular Amplitude fields are turned “Off”	ModeEditorPage ->Parameters tab	F and G (AOO/AAI or VOO/VVI)	F and G (AOO/AAI or VOO/VVI)	Pass
Hysteresis field is editable and “On”	ModeEditorPage ->Parameters tab	H (only AAI and VVI)	H (only AAI and VVI)	Pass
Hysteresis field is editable and “Off”	ModeEditorPage ->Parameters tab	I (only AAI and VVI)	I (only AAI and VVI)	Pass
Rate smooth toggling editing for both Up and Down	ModeEditorPage ->Parameters tab	C and D (Only AAI and VVI)	C and D (Only AAI and VVI)	Pass
Rate smoothing dropdown menu for up and down	ModeEditorPage ->Parameters tab ->Rate smoothing dropdown menu	J and K (only in AAI and VVI)	J and K (only in AAI and VVI)	Pass
Lowest limit for all parameters in each pacing mode	ModeEditorPage ->Parameters tab	L (AAI and VVI also representative of AOO and VOO values)	L (AAI and VVI also representative of AOO and VOO values)	Pass

Upper limit for all parameters in each pacing mode	ModeEditorPage ->Parameters tab	M (AAI and VVI also representative of AOO and VOO values)	M (AAI and VVI also representative of AOO and VOO values)	Pass
Checked steps for each parameter	ModeEditorPage -> Parameters tab	N	N	Pass
Clicked on Summary tab	ModeEditorPage -> Summary tab	O	O	Pass
Clicked on Egram Tab	ModeEditorPage -> Egram tab	P	P	Pass
Clicked on Save Parameters	ModeEditorPage -> Parameters tab	Q	Q	Pass
Clicked on Revert Parameters	ModeEditorPage -> Parameters tab	R	R	Pass

3.5. GenAI Usage

GenAI (ChatGPT) was used at different stages in the development of the DCM. As creating a GUI was an unfamiliar concept, it was initially used to map out, explain and recommend widgets and features from PyQt5 to use for the DCM. It was useful for mapping out the general flow and relation between different Qt features and their sections for the DCM. The idea to use JSON and a hash password was also from ChatGPT and the syntax was verified using it.

Some features relied more on GenAI for their creation, For instance, _report_header_html, _params_table_html, _save_pdf, _export_brady_params_report, and _export_temp_params_report. This was primarily due to the use of HTML - an unfamiliar language, to create the reports. As the requirements regarding the reports were quite unclear, ChatGPT was used to help achieve a more encompassing, acceptable result - exporting the reports and saving them as PDFs. It was also used for the same purpose in _refresh_summary to create the Summary tab using HTML. Additionally, it was also used for clamping logic and the lambda based ValueError functions for clamping values within range.

ChatGPT was also consulted for class D1EgramView to help simulate values for the Egram, and learn how to use the Painting imports from PyQt. GenAI was consulted to advise on and confirm the specific details on how to use the features to achieve the desired outcome in the DCM.

Overall, ChatGPT was also used throughout the coding process to check for and fix errors to ensure a functional, running code.

4. Appendix

4.1. Appendix 1. AOO, VOO, AAI, and VVI Test Results

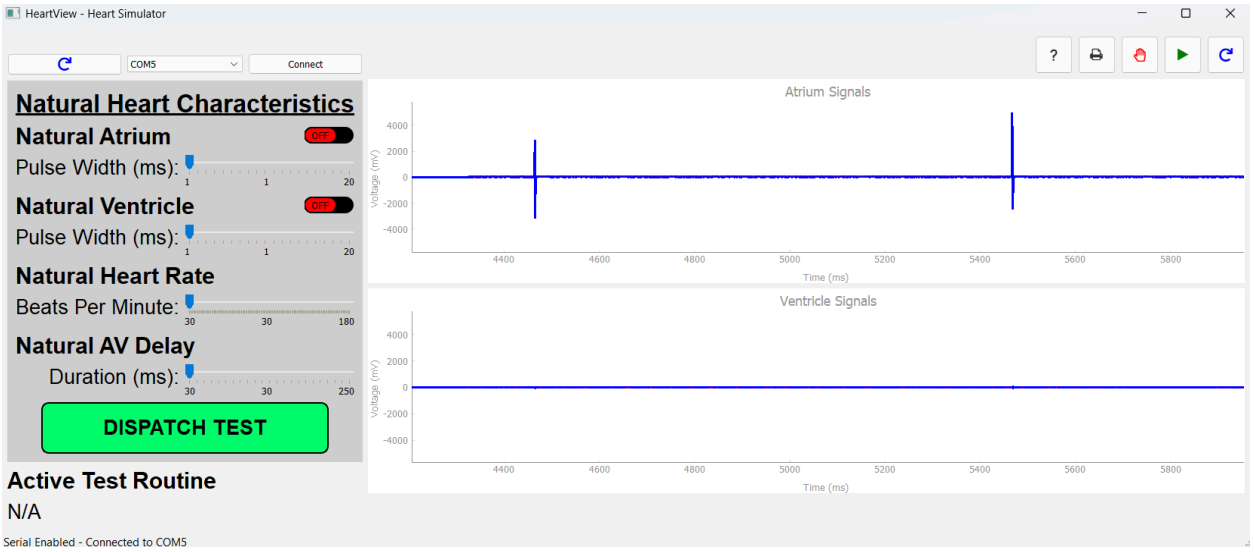

Figure 1. AOO Default

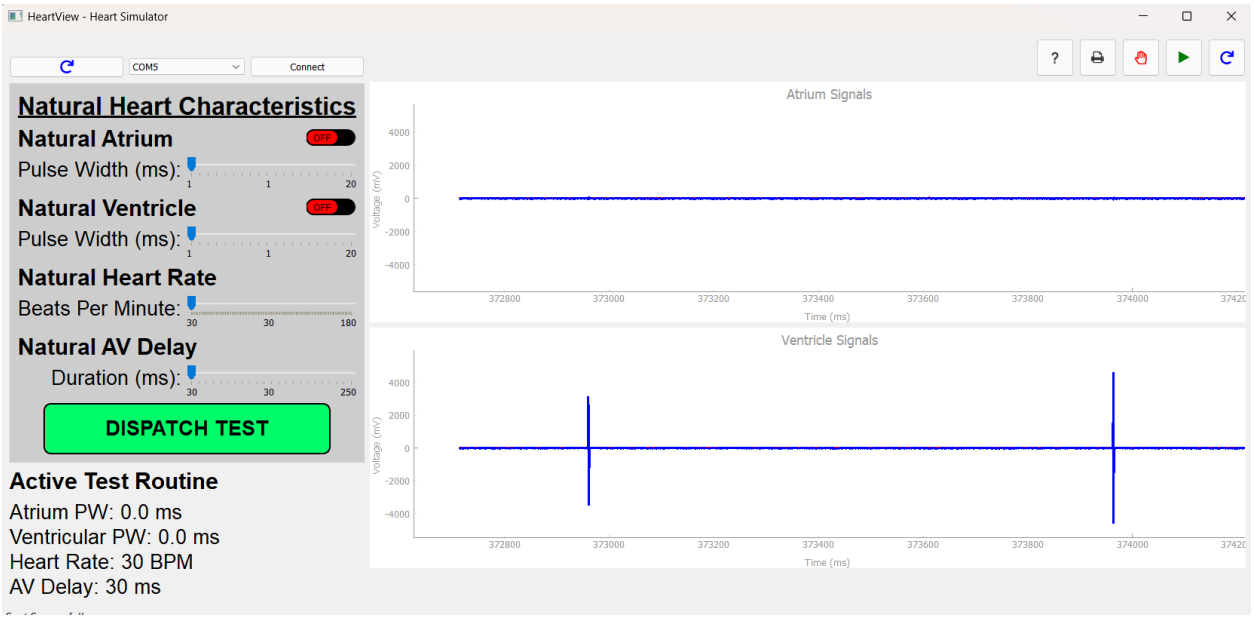

Figure 2. VOO Default

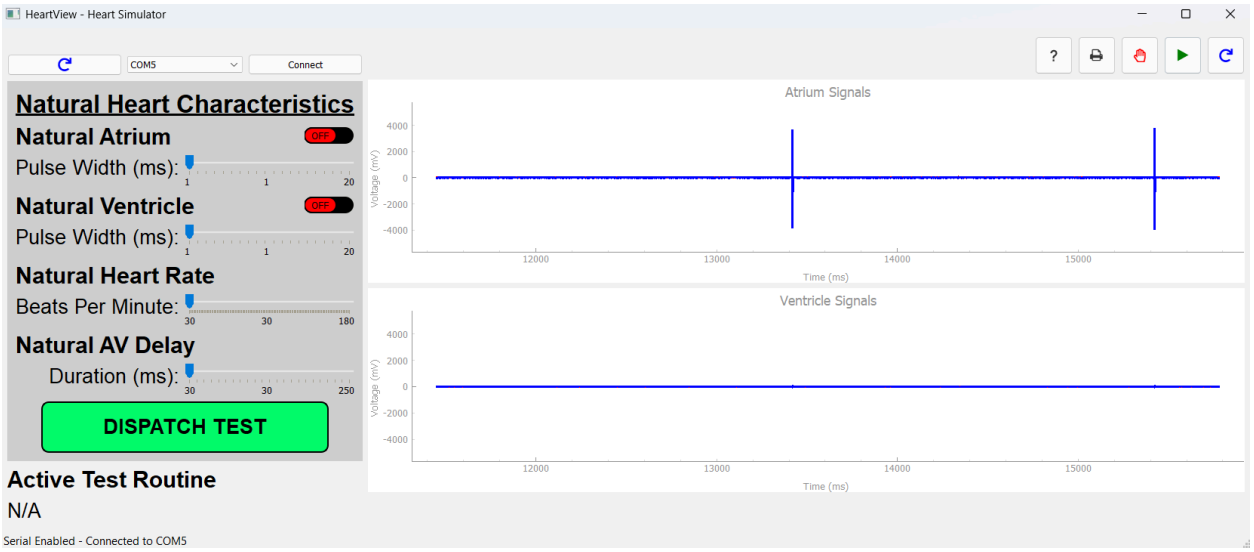

Figure 3. AOO Interval 2000

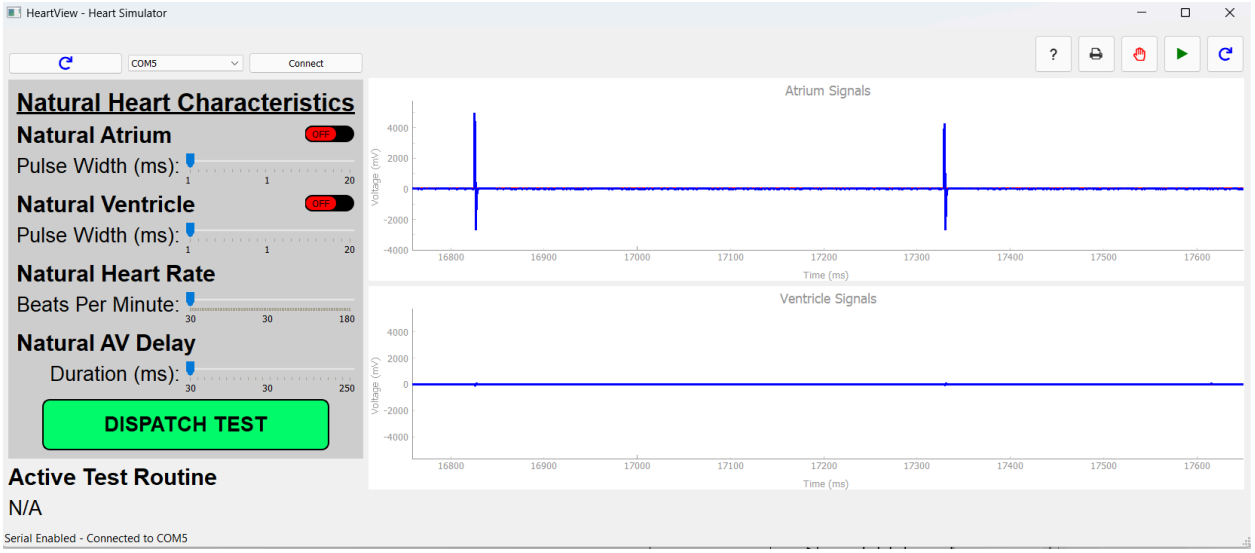

Figure 4. AOO Interval 500

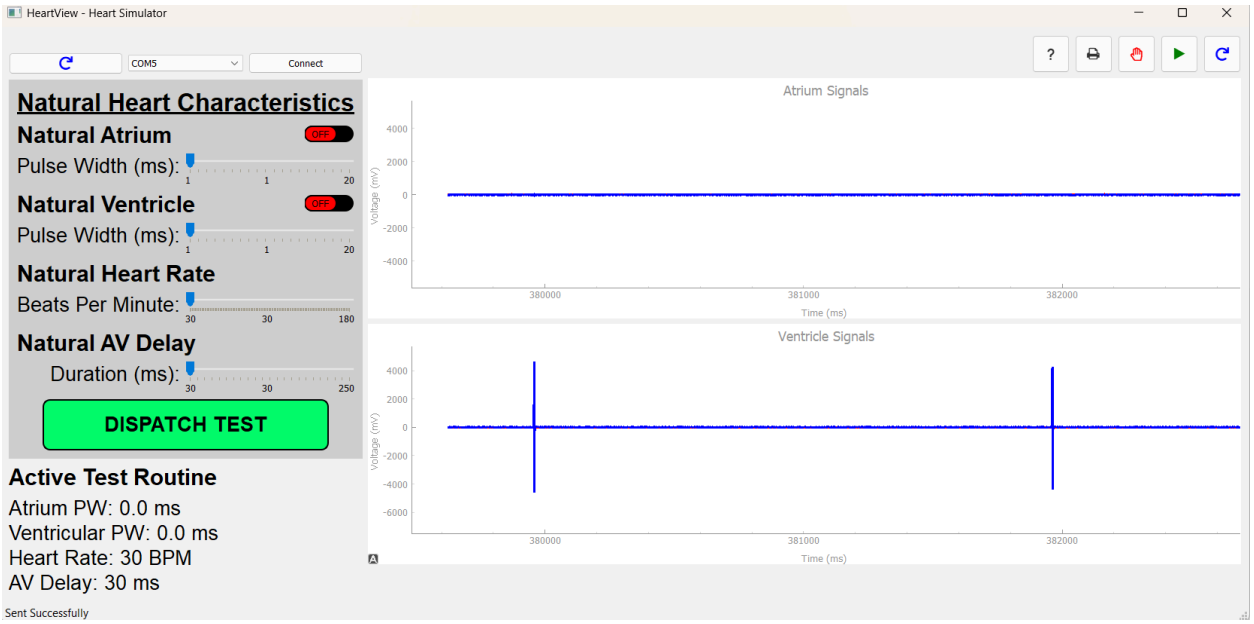

Figure 5. VOO Interval 2000

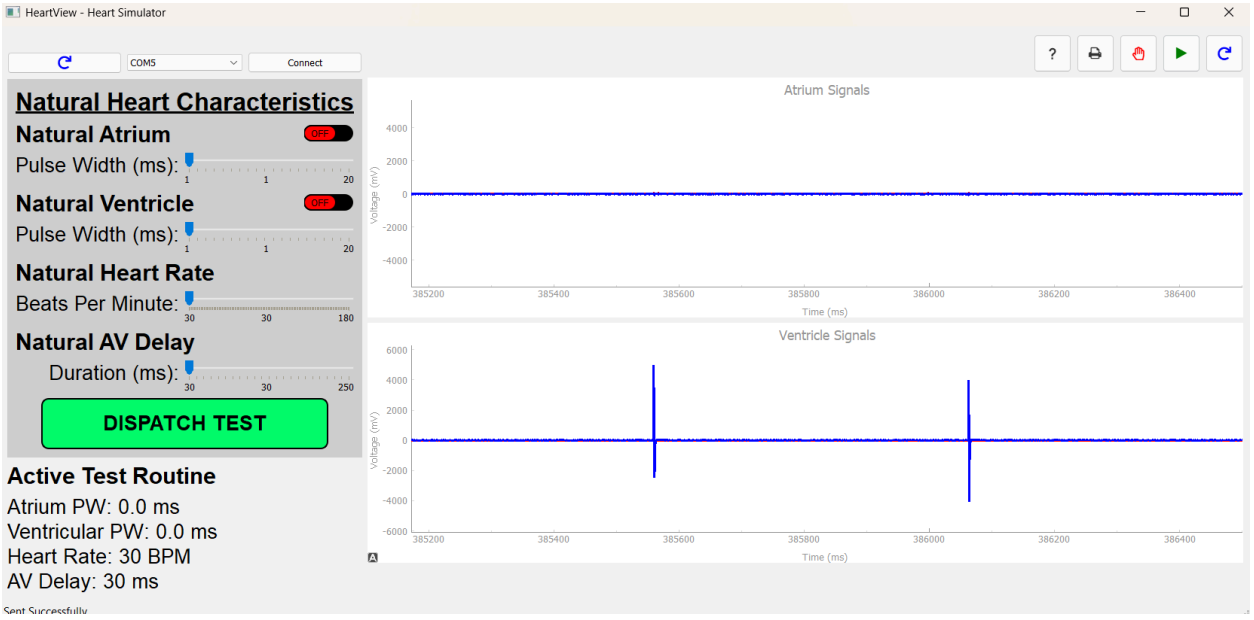

Figure 6. VOO Interval 500

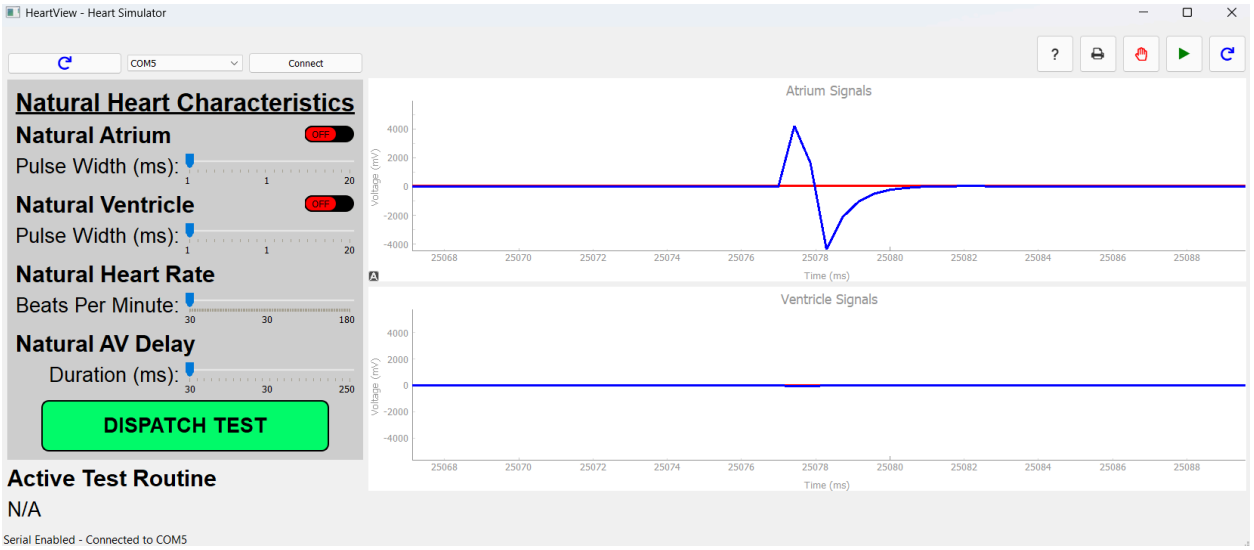

Figure 7. AOO Width 0.2

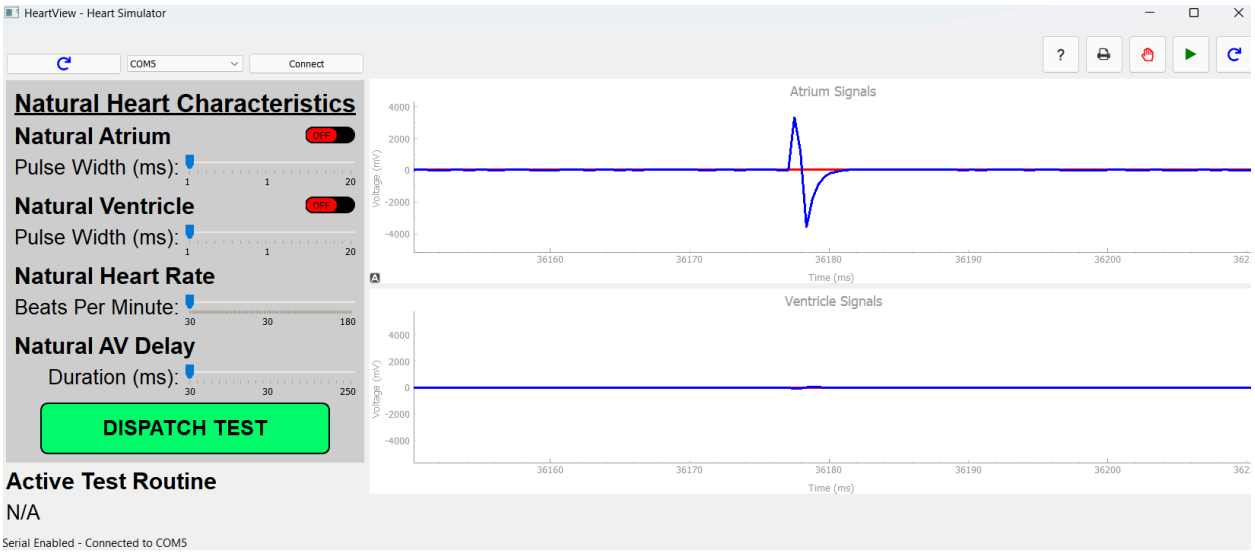

Figure 8. AOO Width 0.8

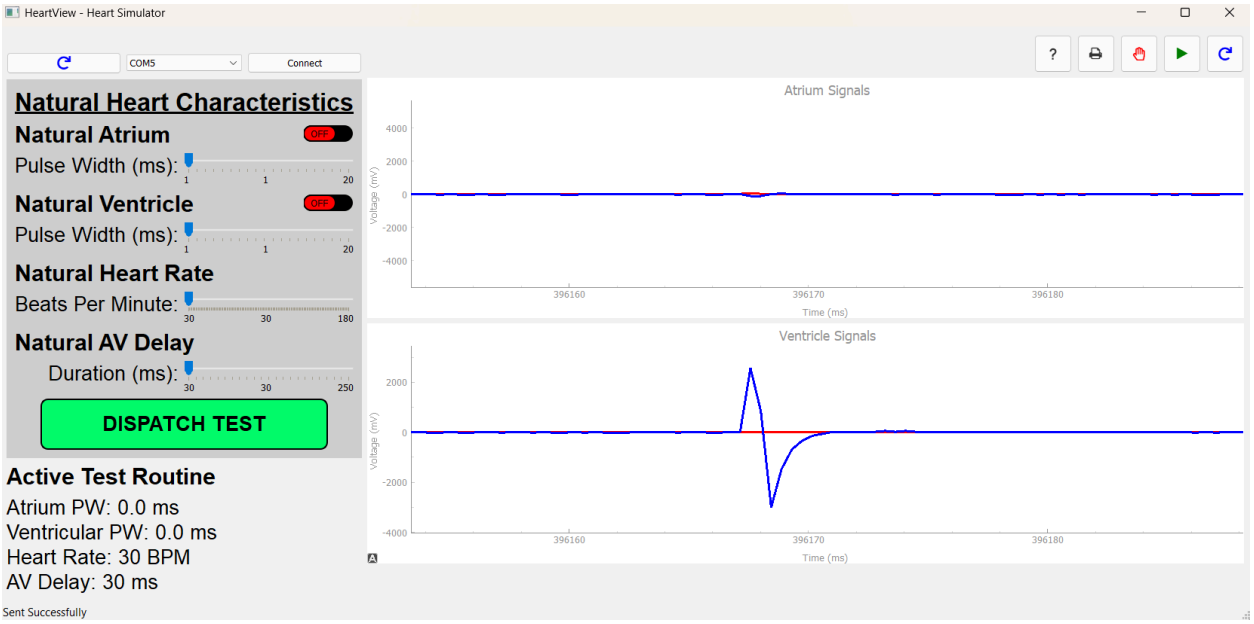

Figure 9. VOO Width 0.2

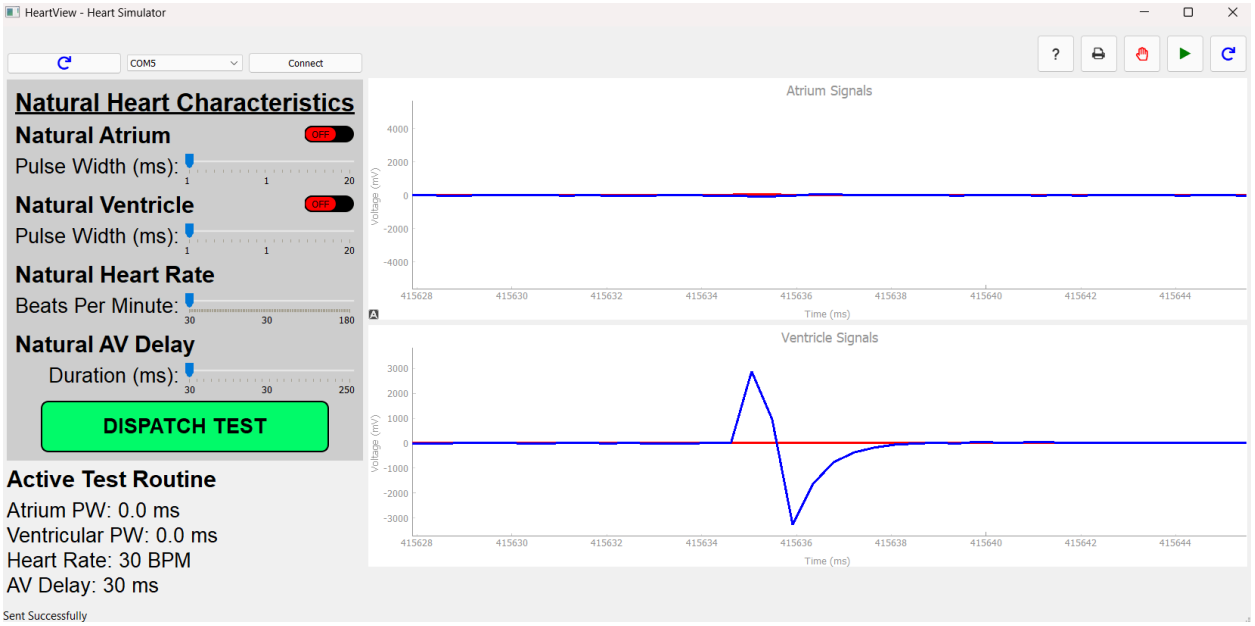

Figure 10. VOO Width 0.8

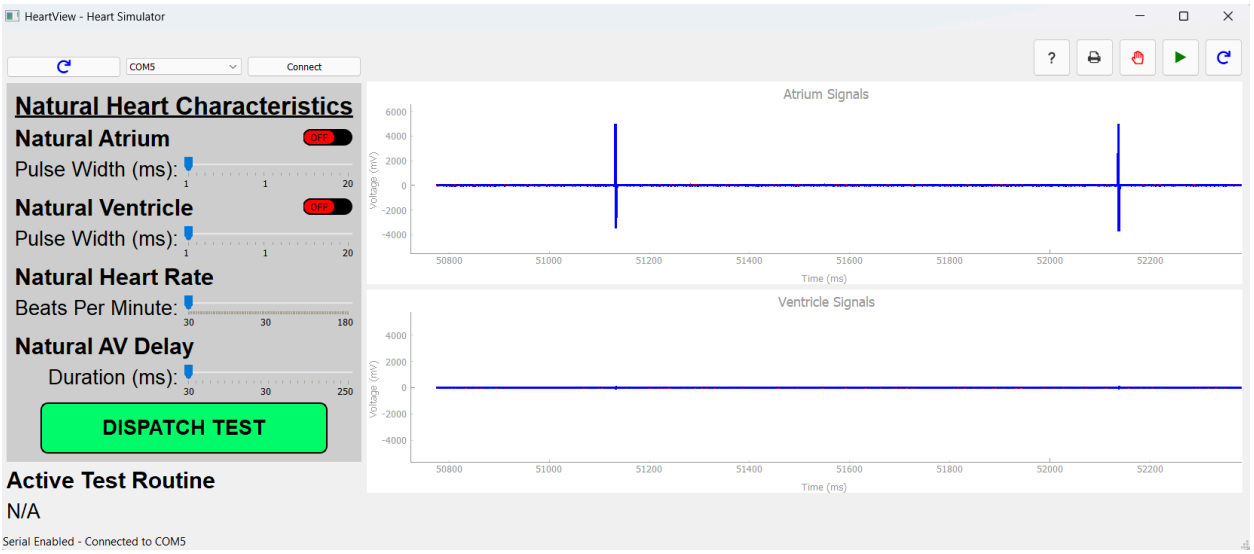

Figure 11. AOO Amp 5000

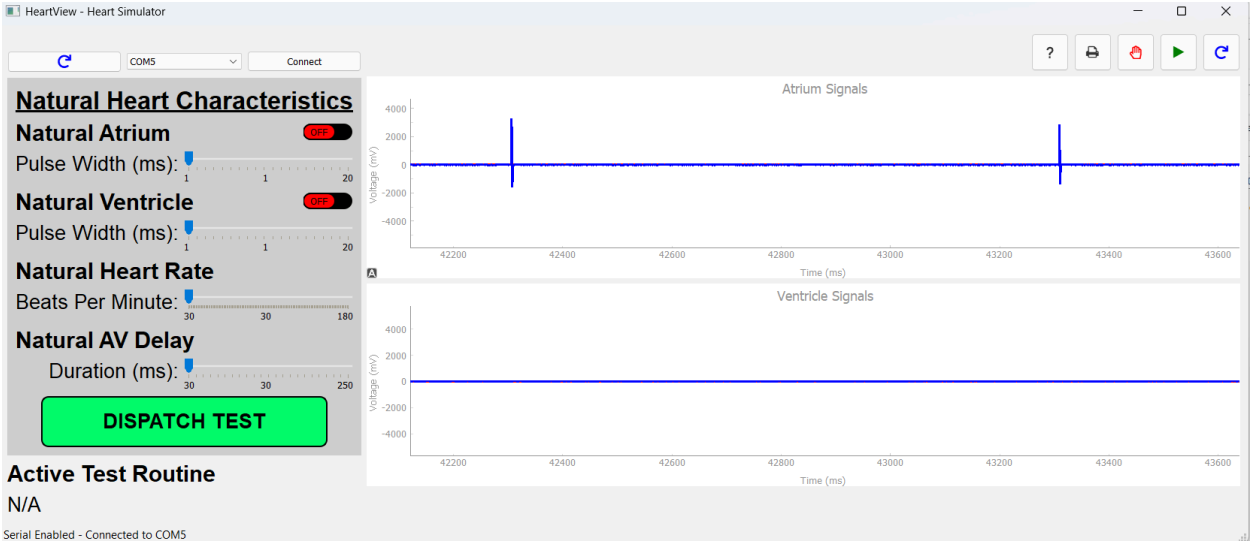

Figure 12. AOO Amp 2000

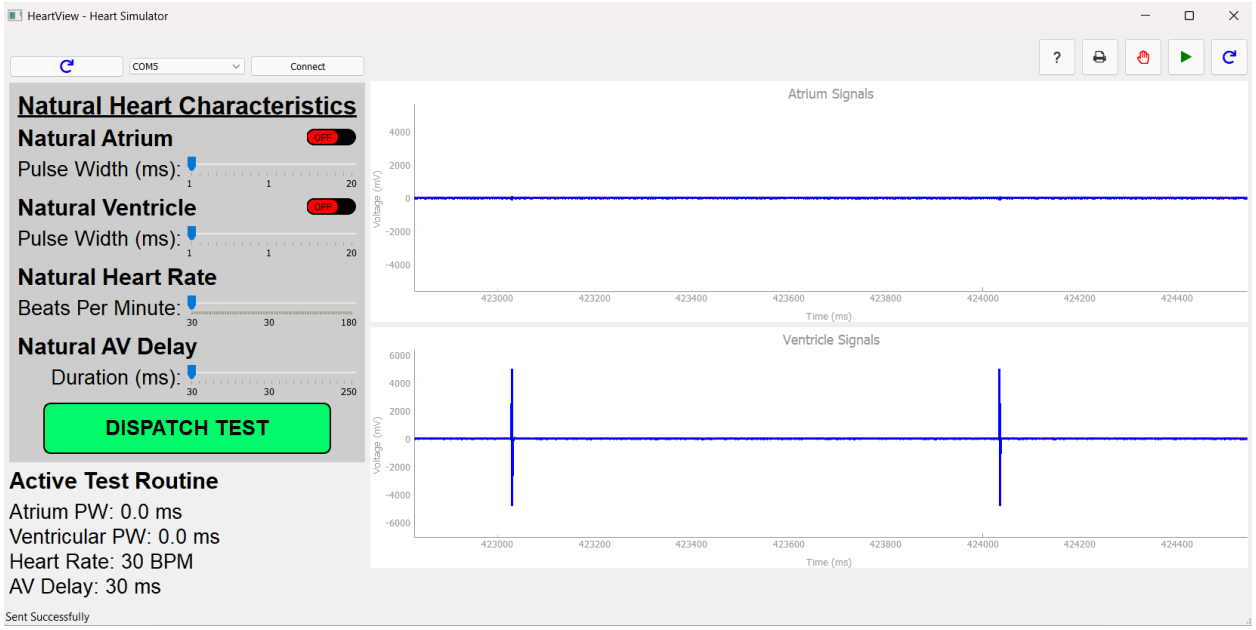

Figure 13. VOO Amp 5000

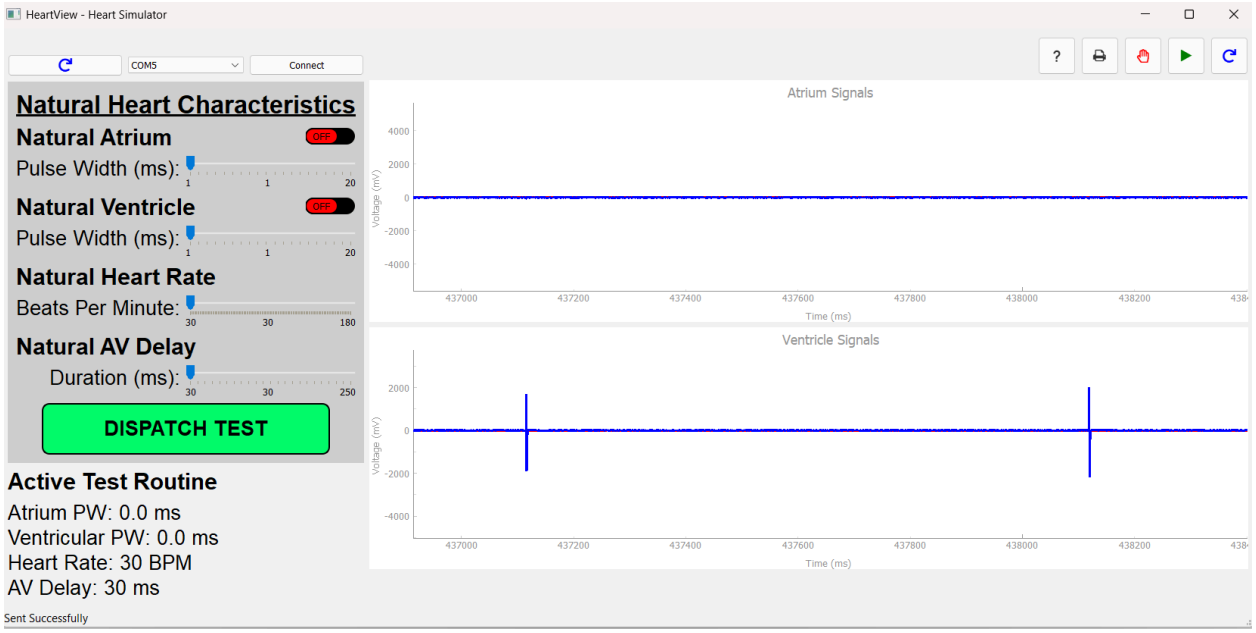

Figure 14. VOO Amp 2000

Figure 15. AAI Default

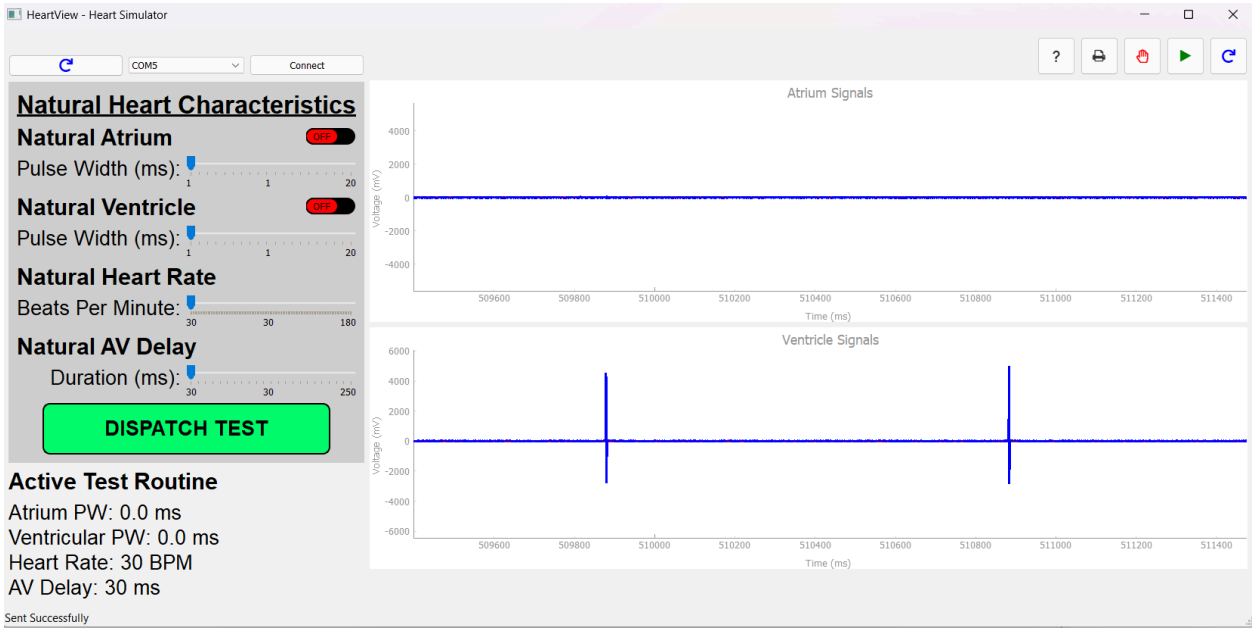

Figure 16. VVI Default

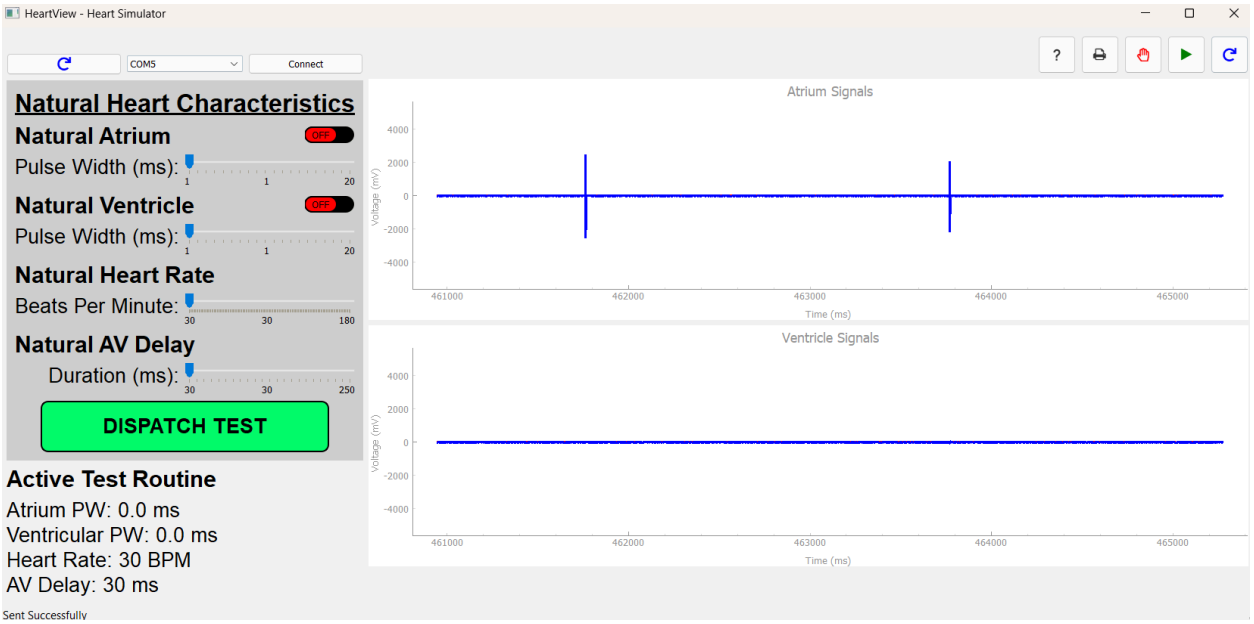

Figure 17. AAI Interval 2000

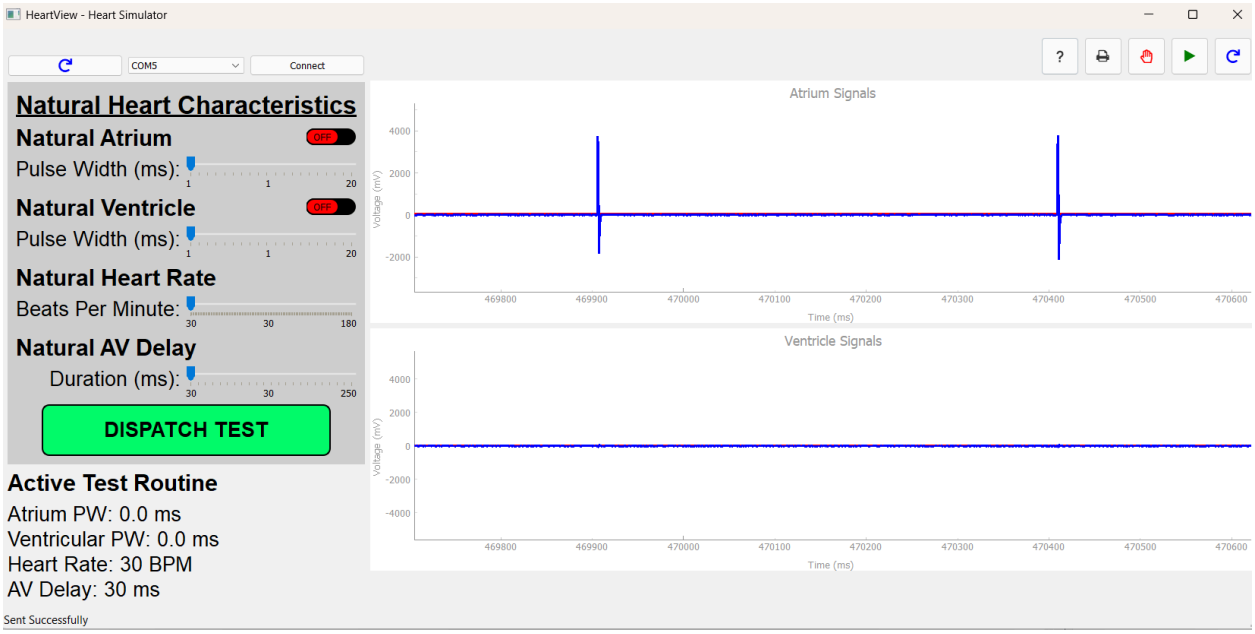

Figure 18. AAI Interval 500

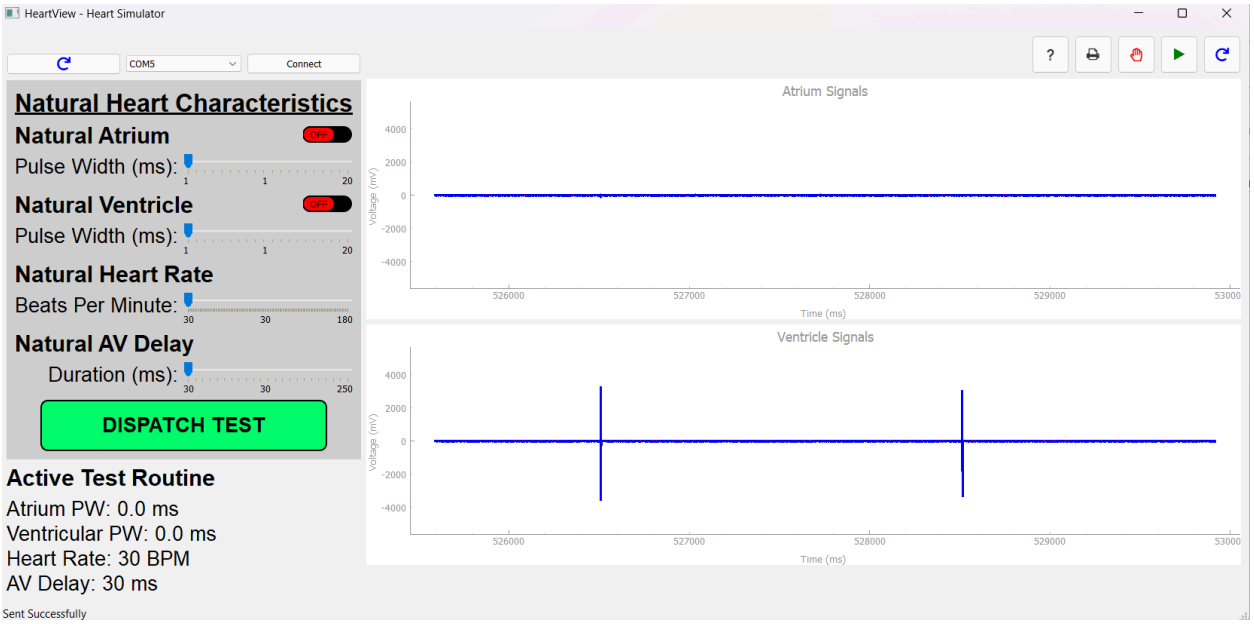

Figure 19. VVI Interval 2000

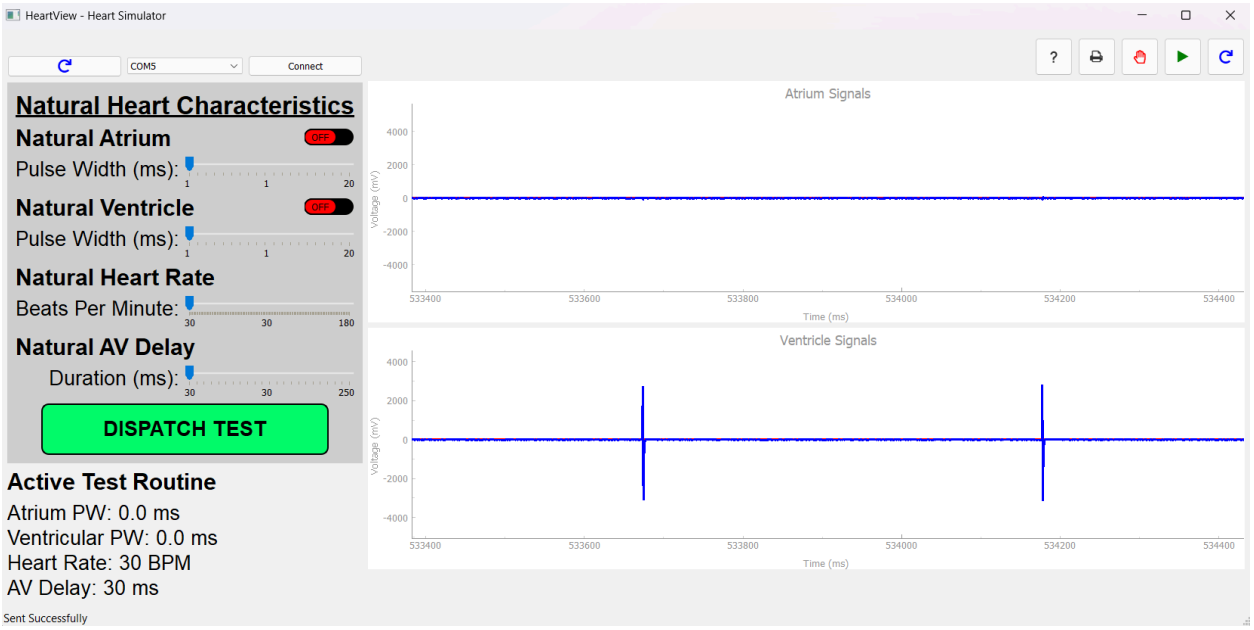

Figure 20. VVI Interval 500

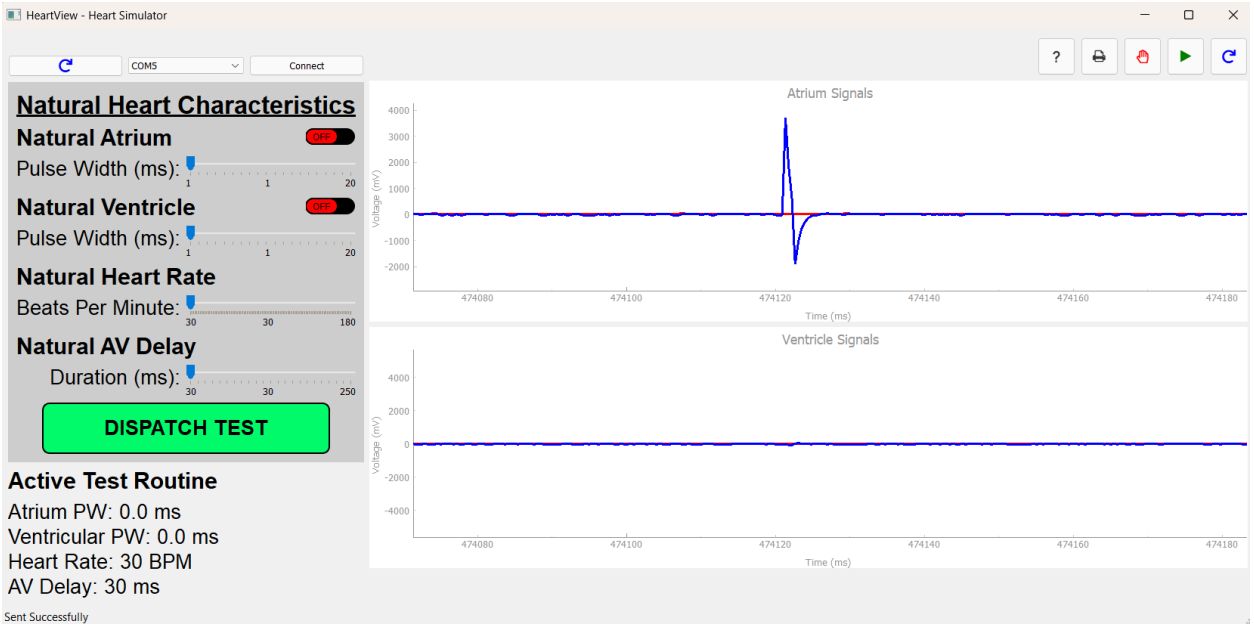

Figure 21. AAI Width 0.2

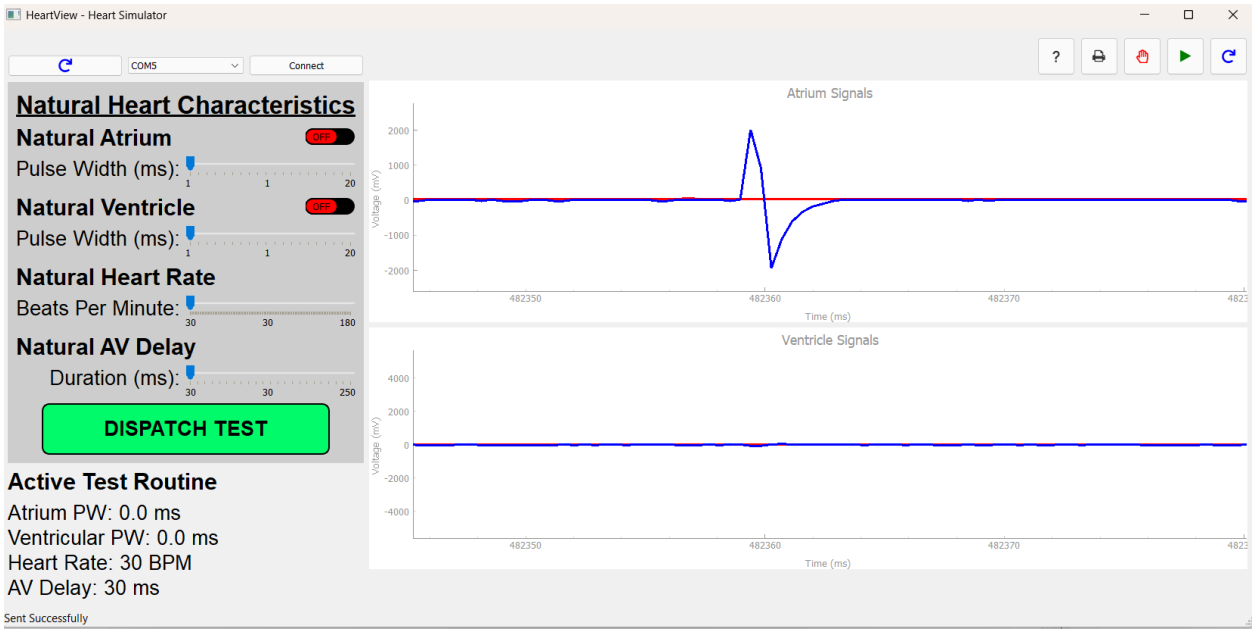

Figure 22. AAI Width 0.8

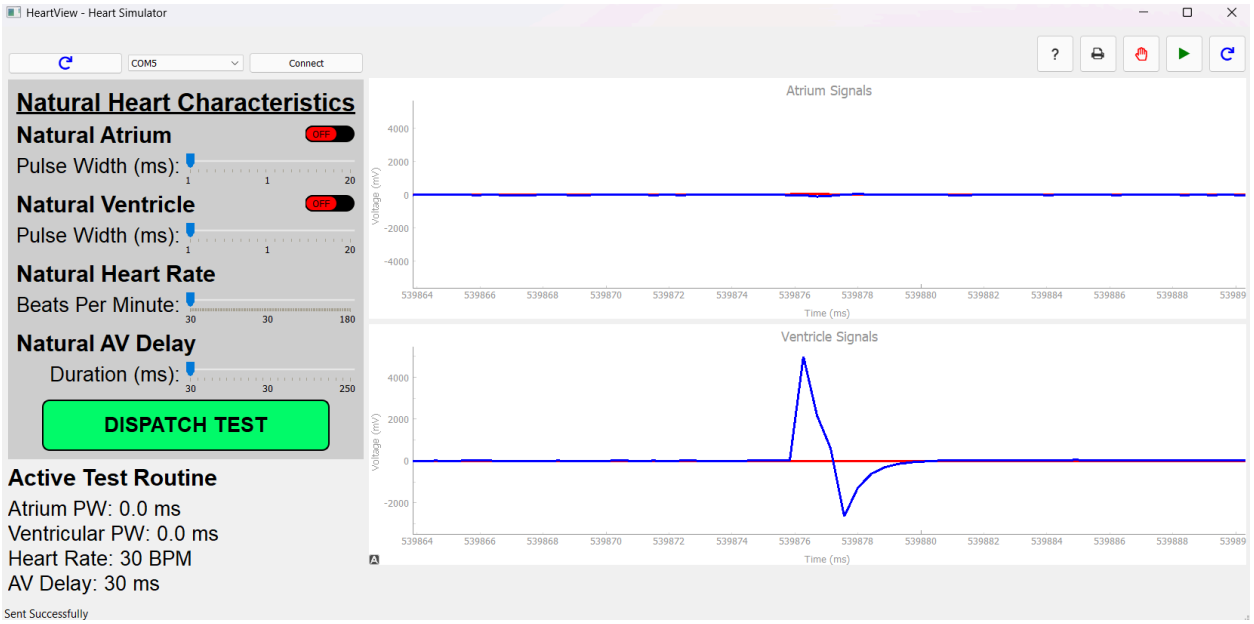

Figure 23. VVI Width 0.2

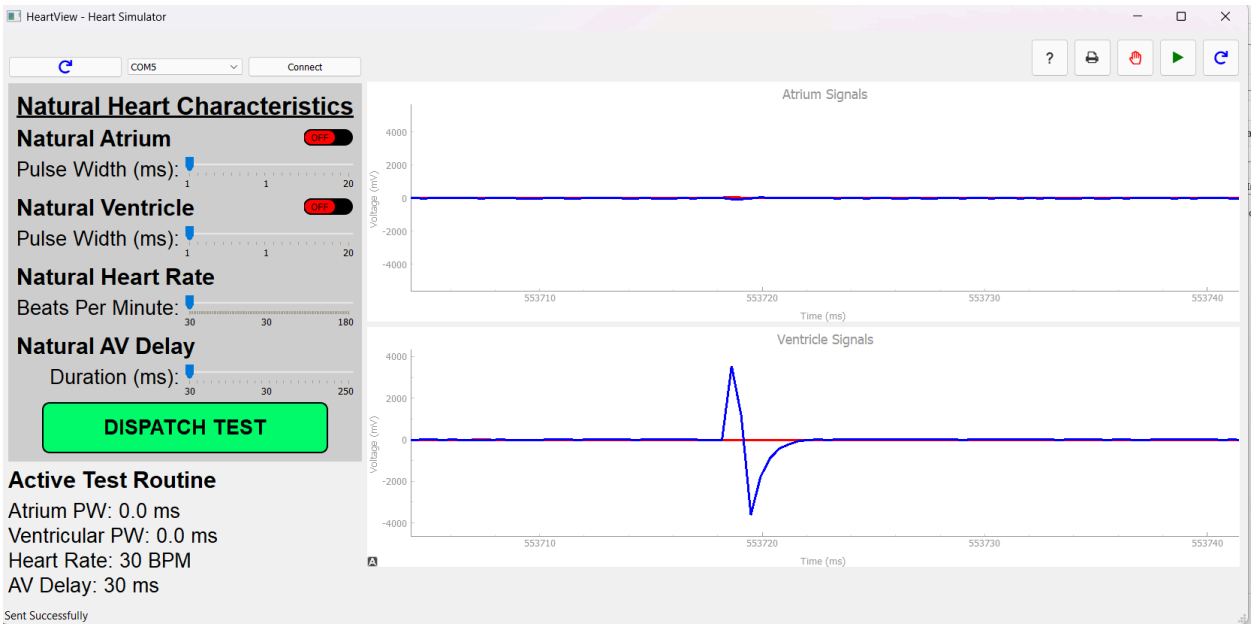

Figure 24. VVI Width 0.8

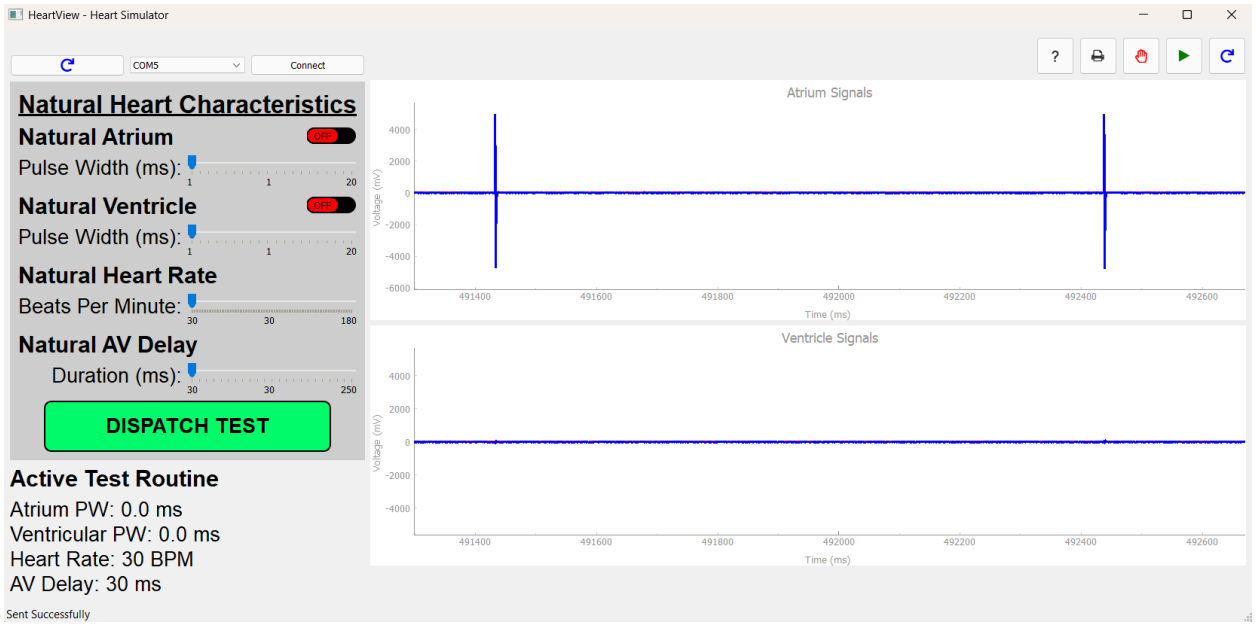

Figure 25. AAI Amp 5000

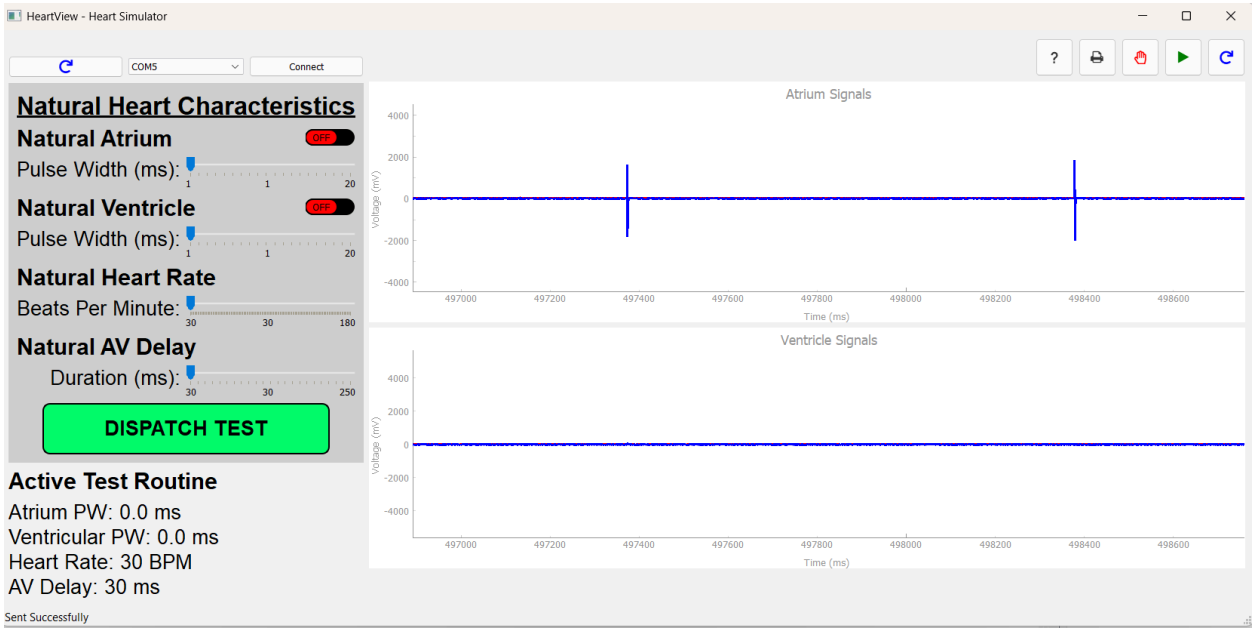

Figure 26. AAI Amp 2000

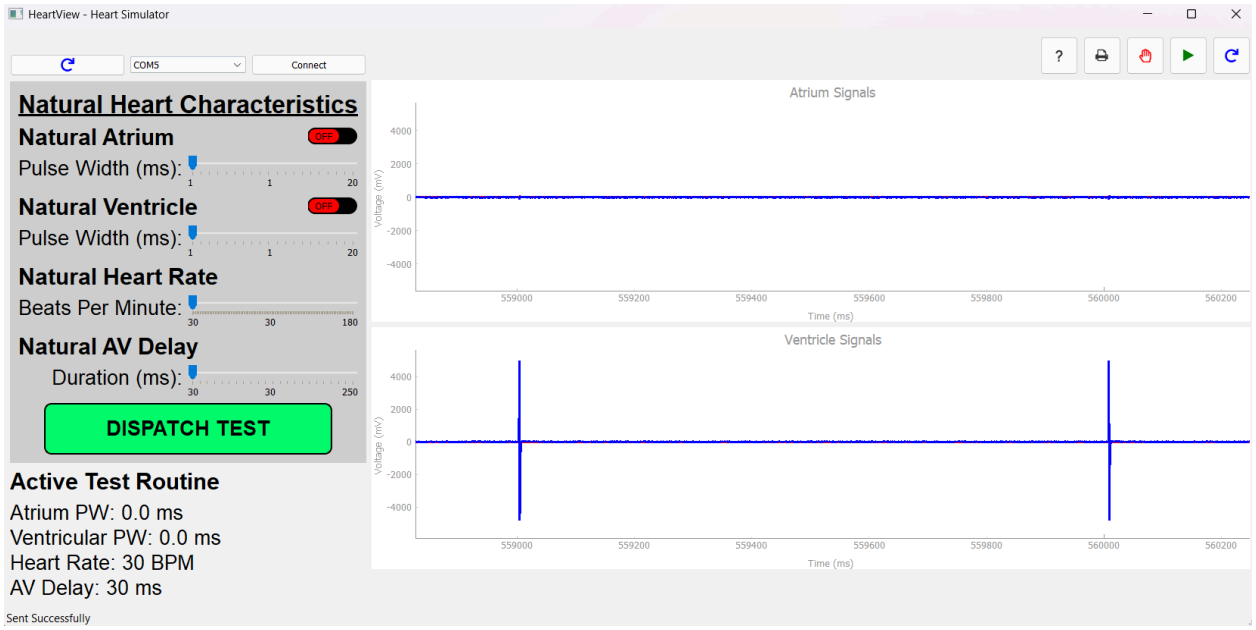

Figure 27. VVI Amp 5000

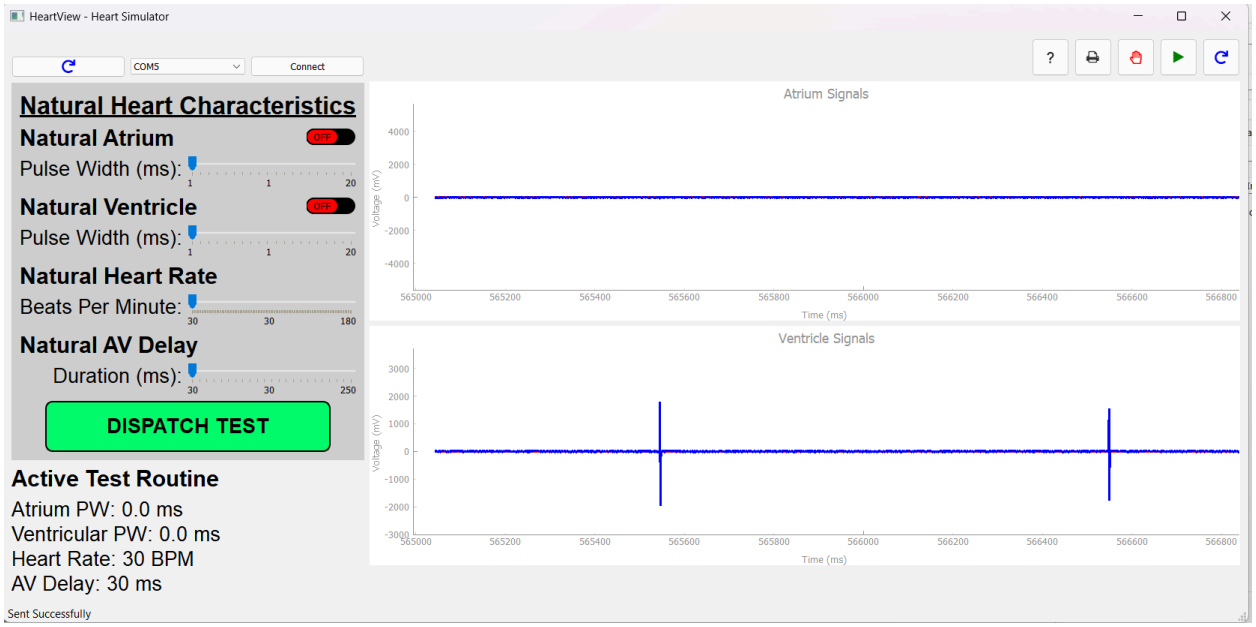

Figure 28. VVI Amp 2000

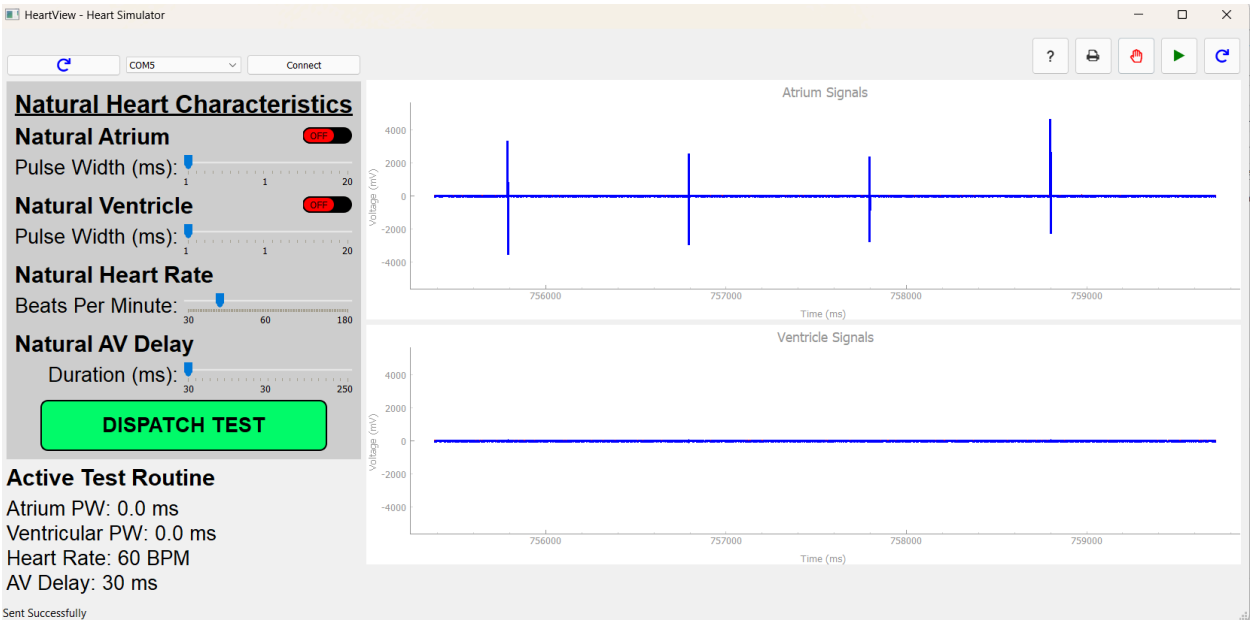

Figure 29. AAI m_as is 0 (constant)

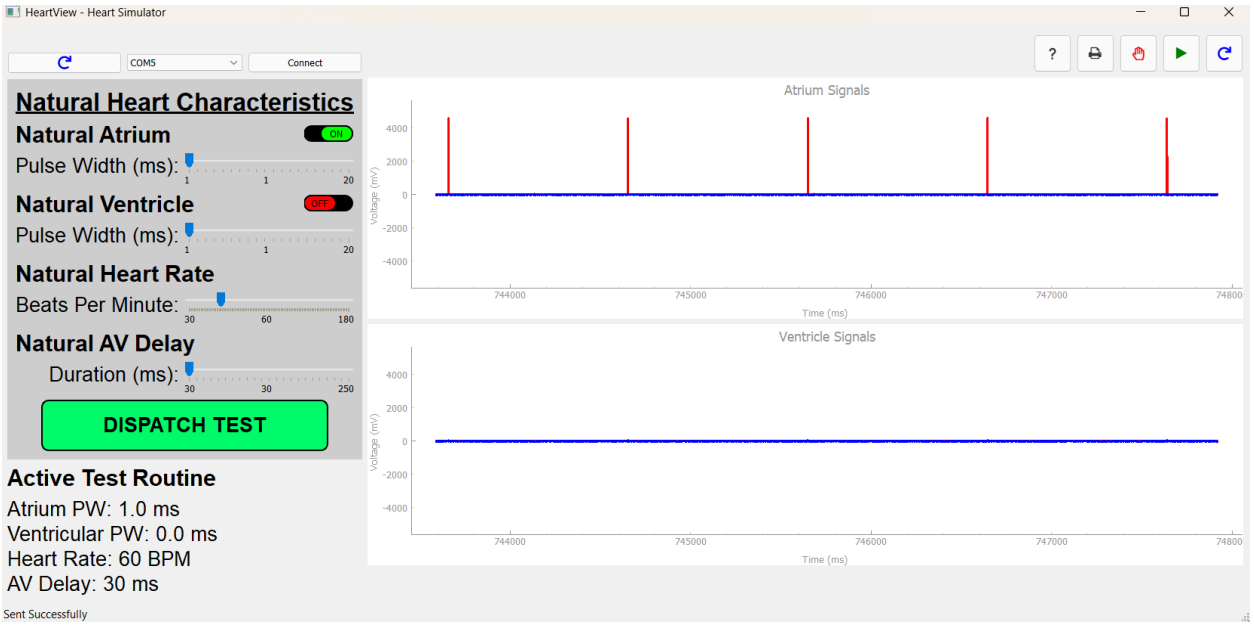

Figure 30. AAI m_as is 1 (constant)

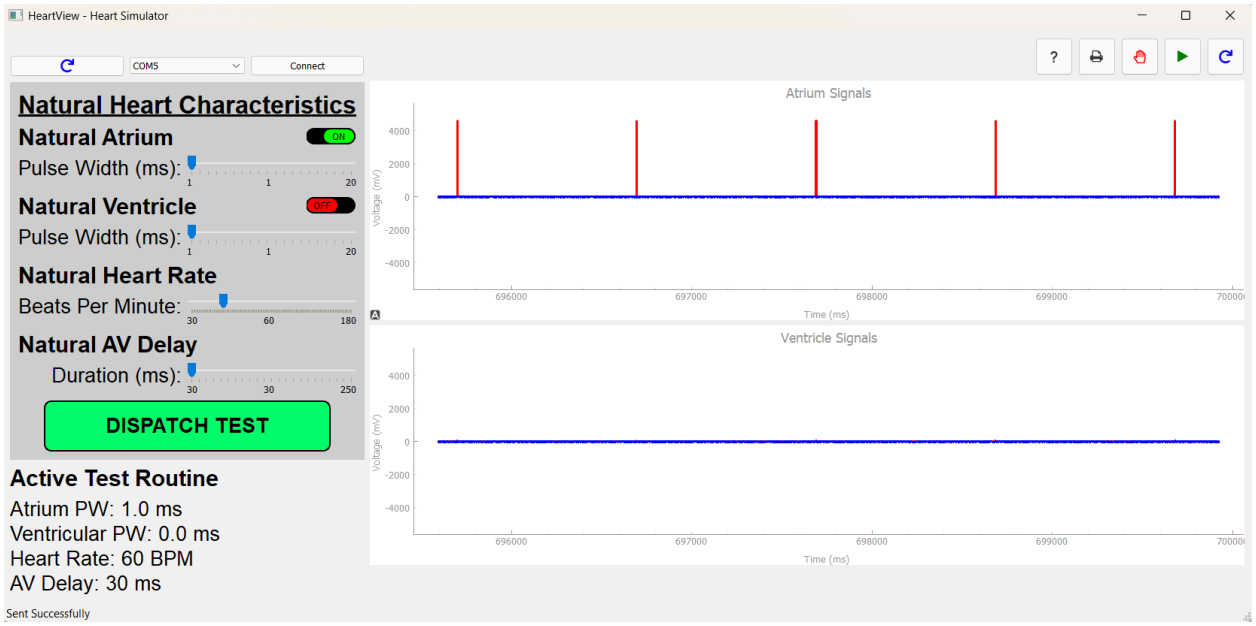

Figure 31. AAI m_as detects 60 bpm Heart Rate (HR)

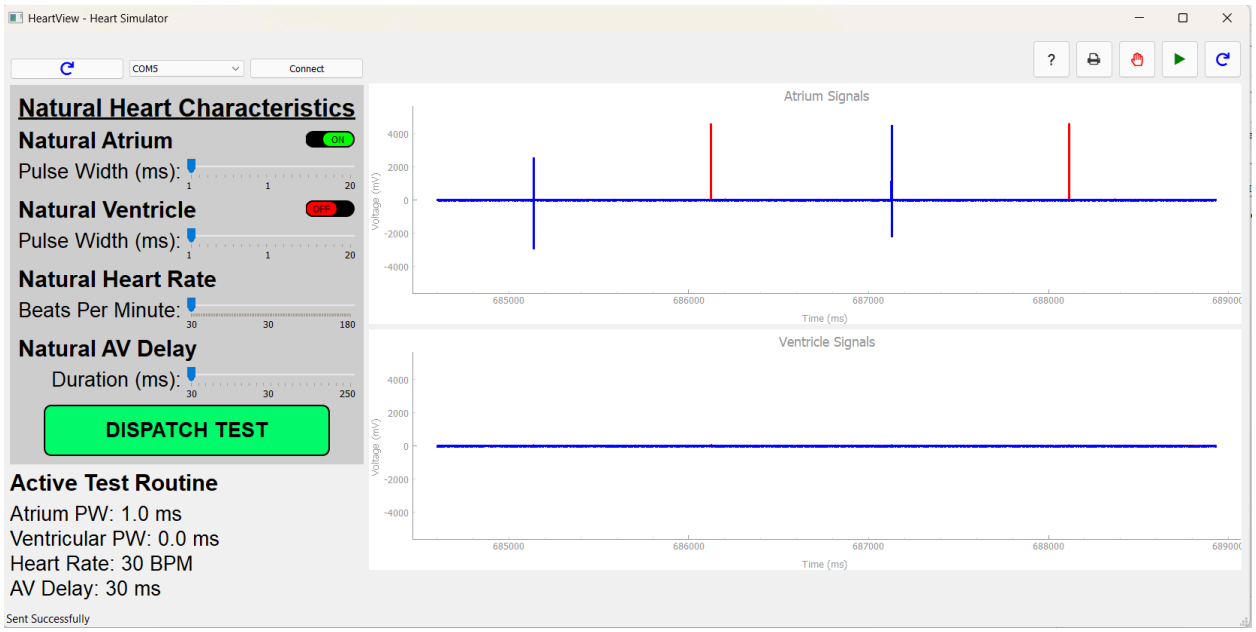

Figure 32. AAI m_as detects 60 bpm Heart Rate (HR)

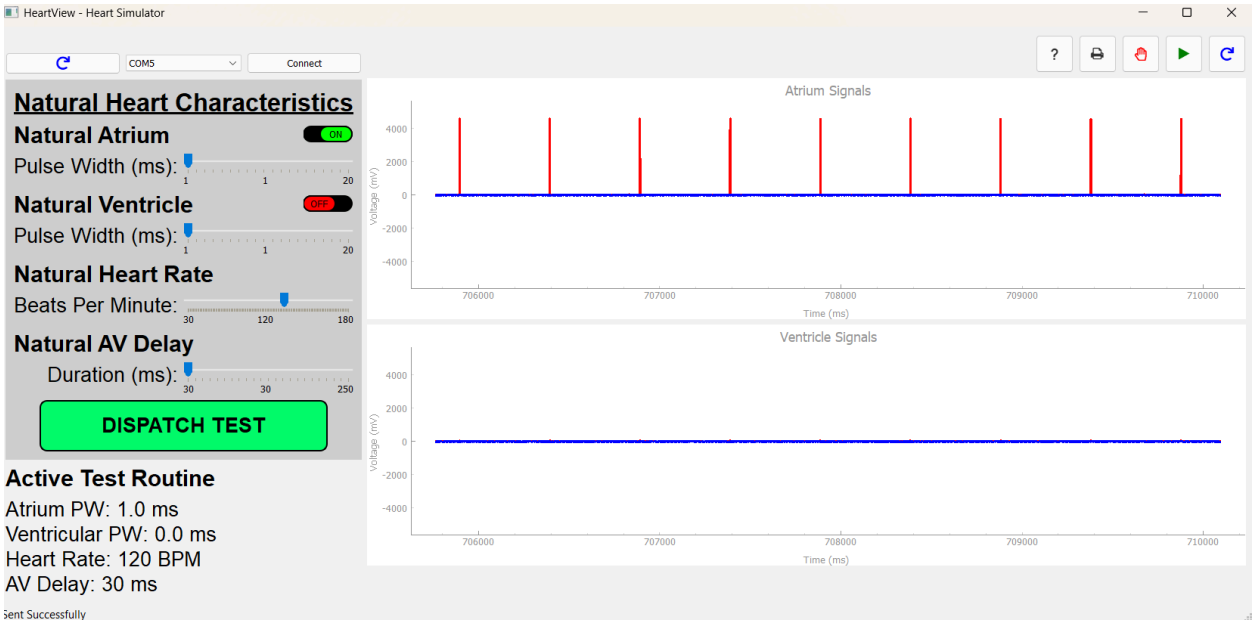

Figure 33. AAI m_as detects 120 bpm Heart Rate (HR)

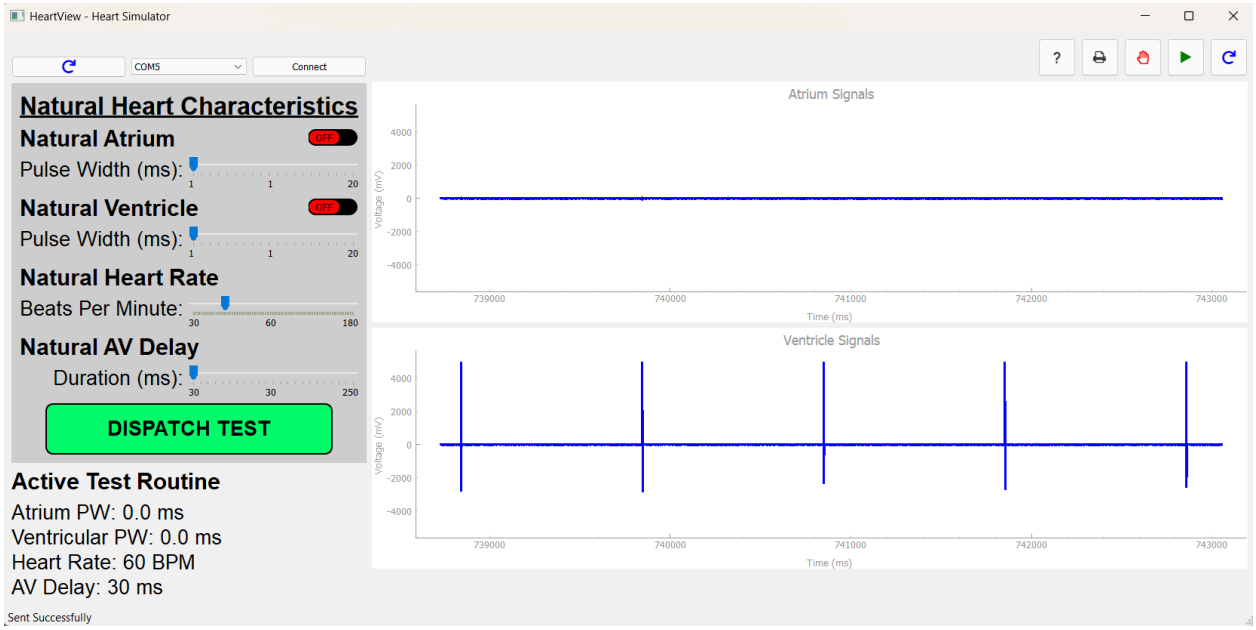

Figure 34. VVI m_as is 0 (constant)

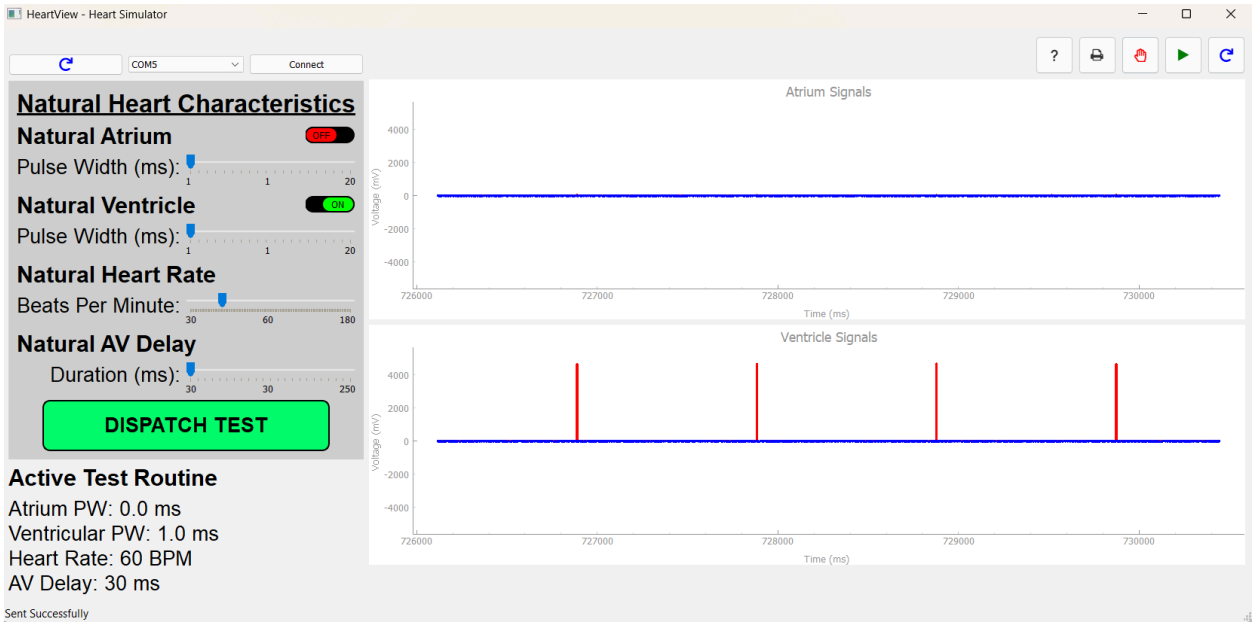

Figure 35. VVI m_as is 1 (constant)

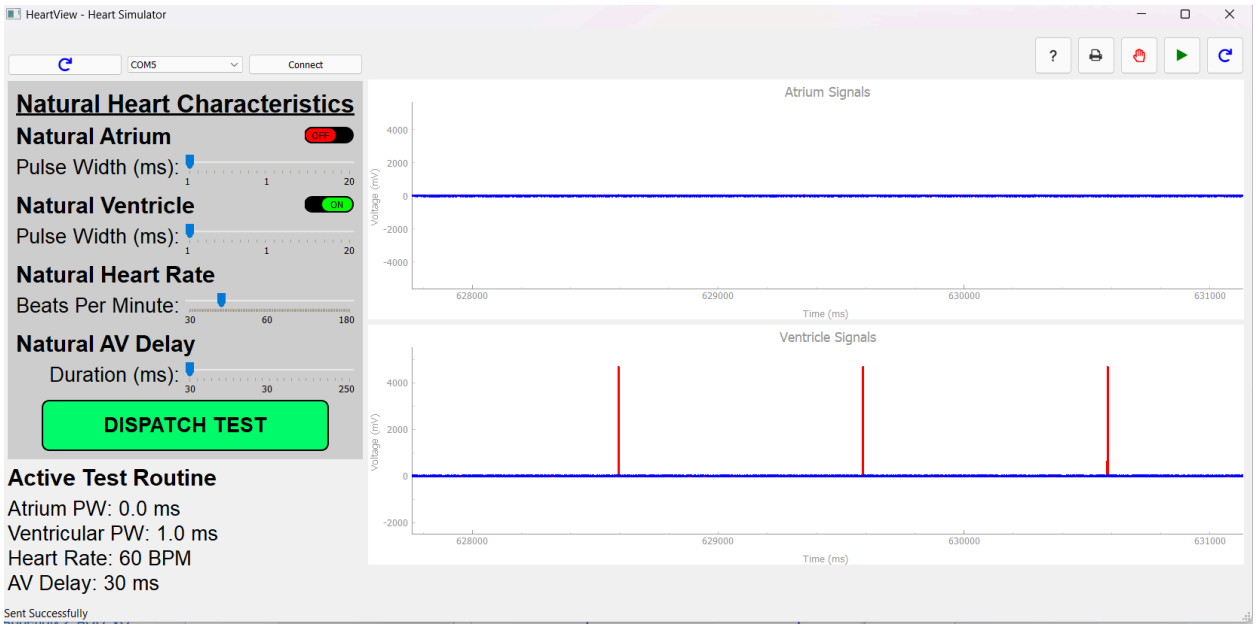

Figure 36. VVI m_as detects 60 bpm Heart Rate (HR)

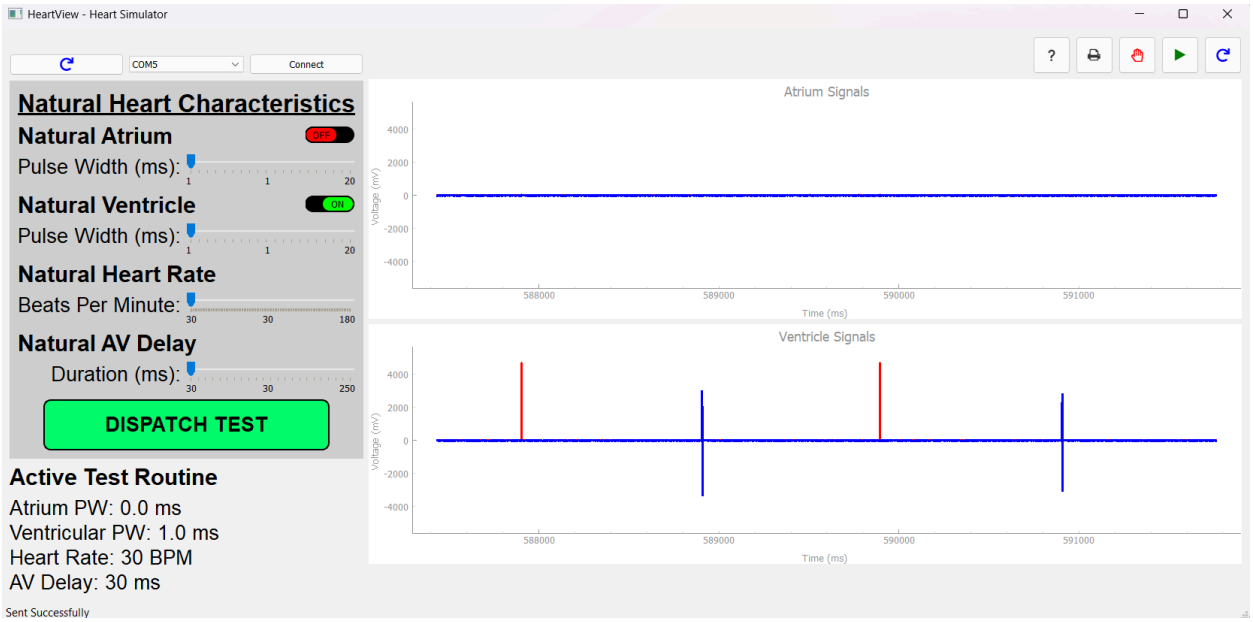

Figure 37. VVI m_as detects 30 bpm Heart Rate (HR)

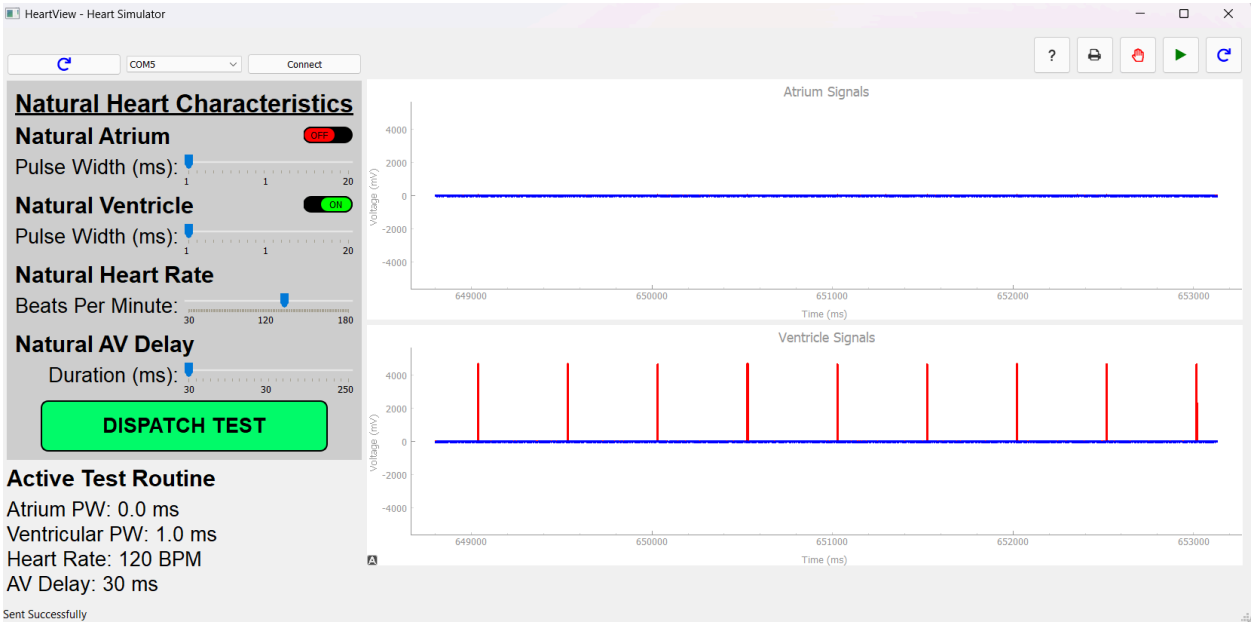

Figure 38. VVI m_as detects 120 bpm Heart Rate (HR)

4.2. Appendix 2. DCM Test Results

DCM Registration test

Registration test with output A (“Please enter a name and password” dialog):

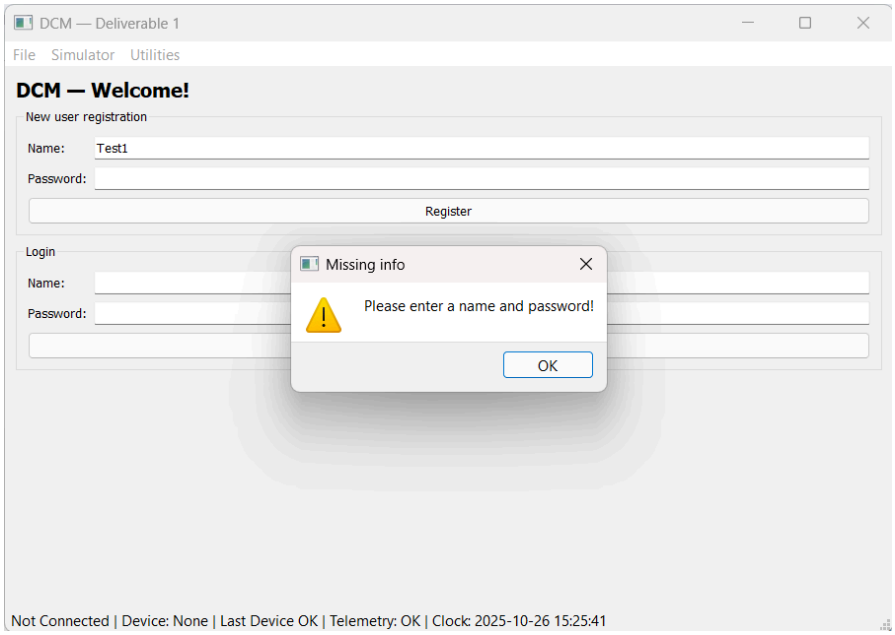

Figure 1: Registration test with output A

Registration test with output B (“User exists or capacity reached” dialog):

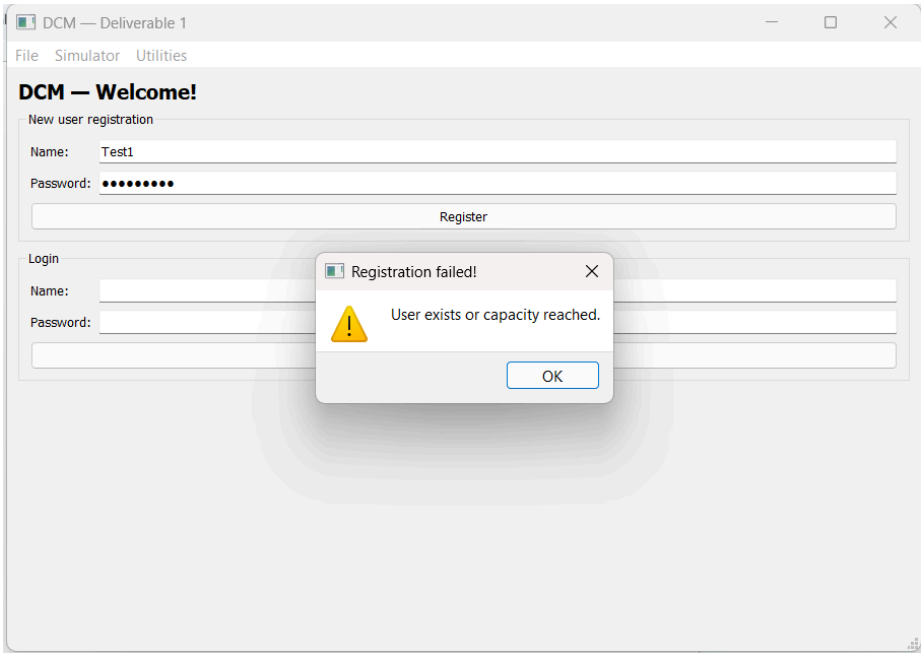

Figure 2: Registration test with output B

Registration test with output C(“Registration complete” dialog):

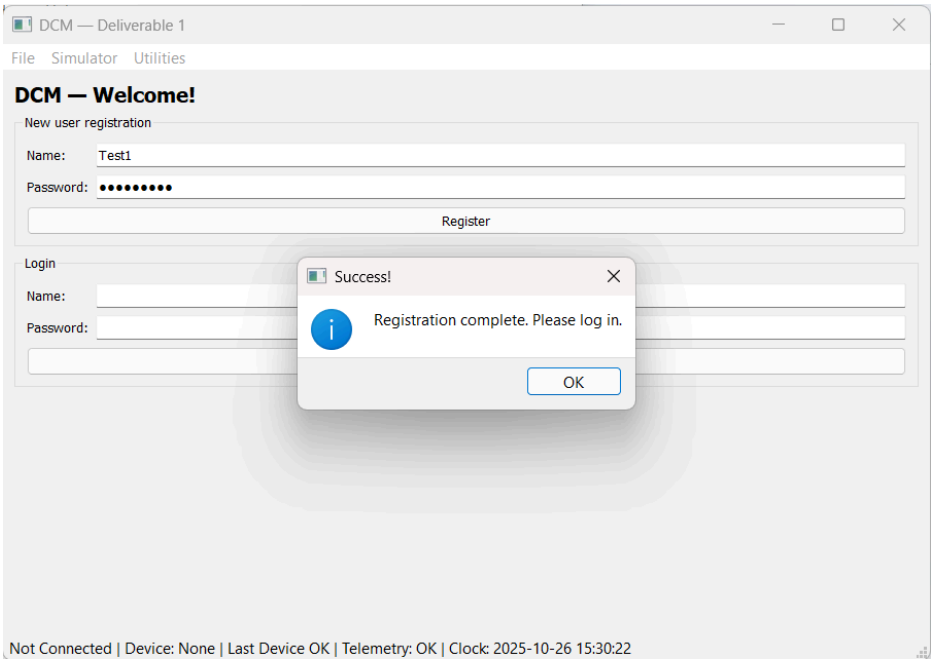

Figure 3: Registration test with output C

Registration test with output D(“Maximum of 10 users stored” dialog):

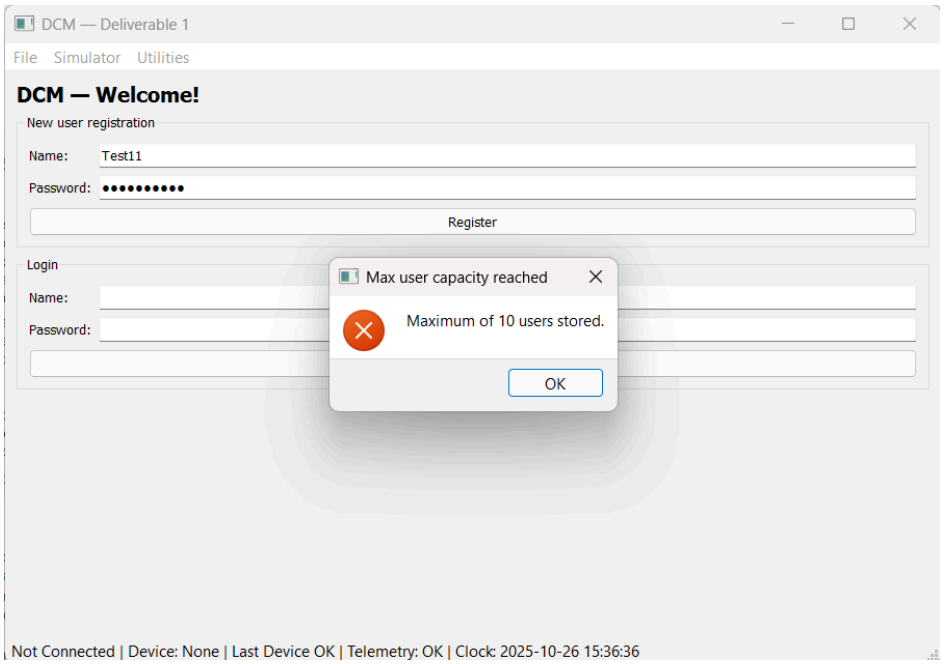

Figure 4: Registration test with output D

DCM Login test

Login test with output A(“Invalid username or password” dialog):

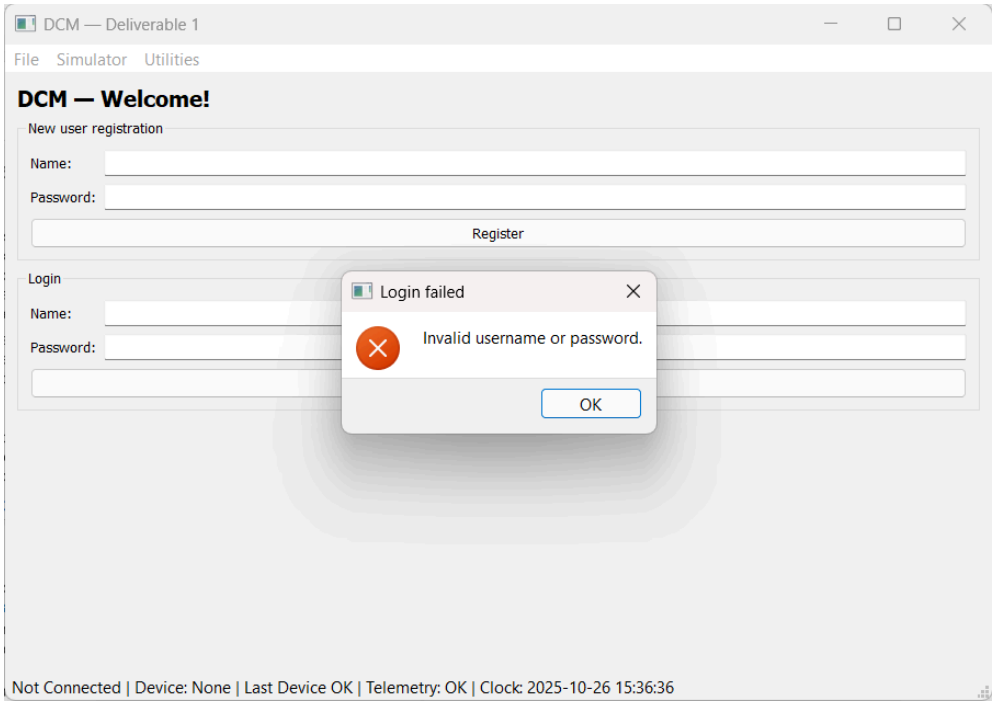

Figure 5: Login test with output A

Login test with output B (Changes from LoginPage to the Dashboard Page with “Logged in as <Name> shown at the bottom left):

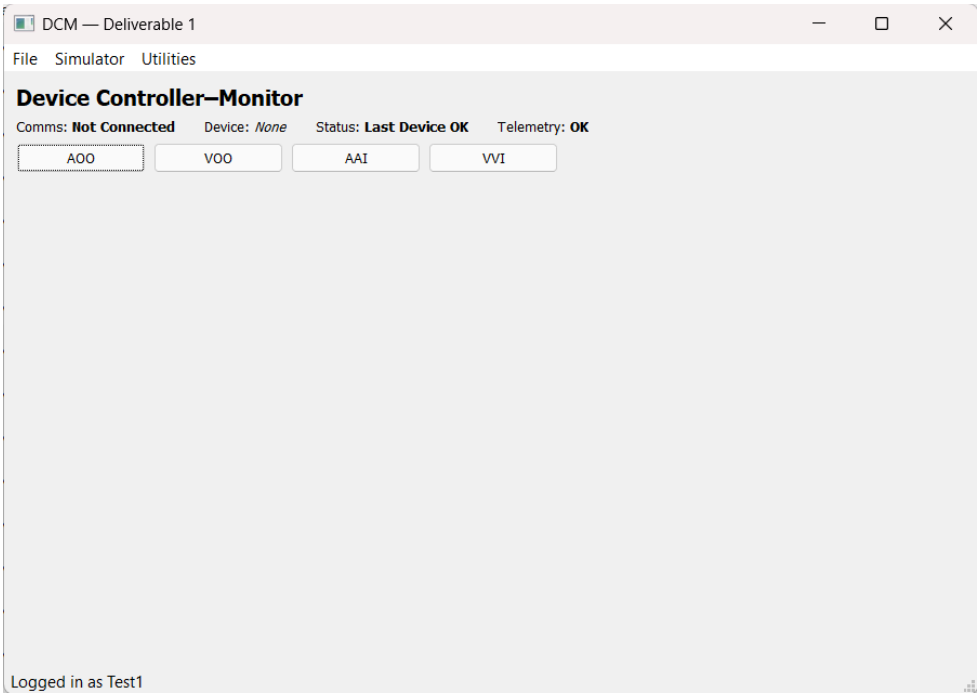

Figure 6: Login test with output B

DCM File dropdown menu test

File test with output A (“Please log in and open the Mode Editor first” dialog):

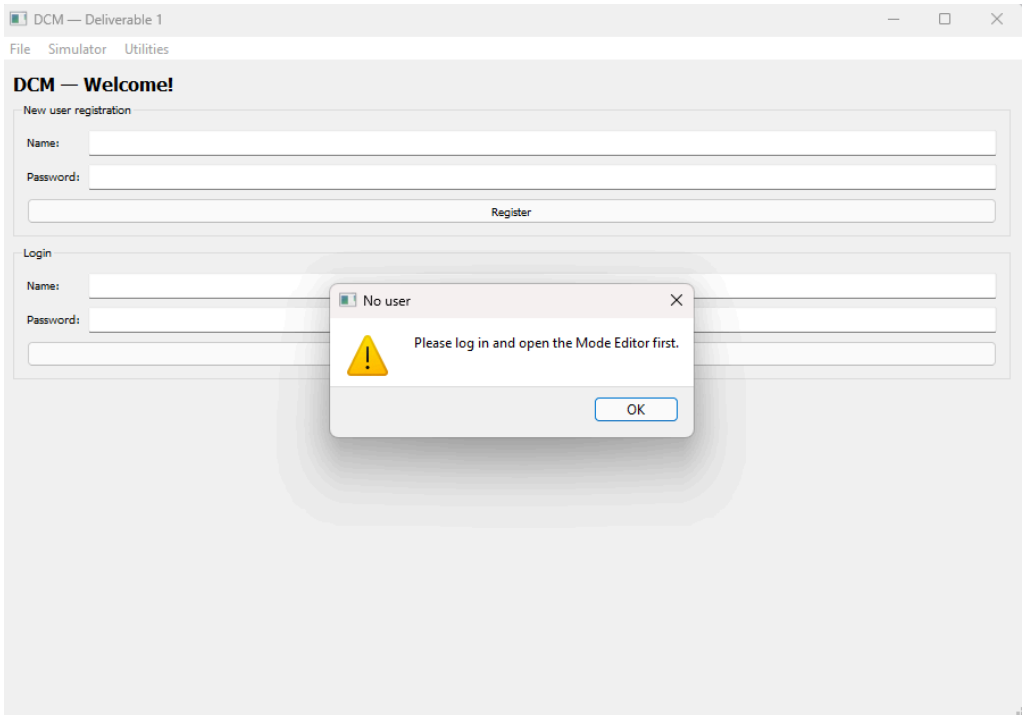

Figure 7: File test with output A

File test with output B (Opens files and allows for the JSON file “dcm_params” to be downloaded):

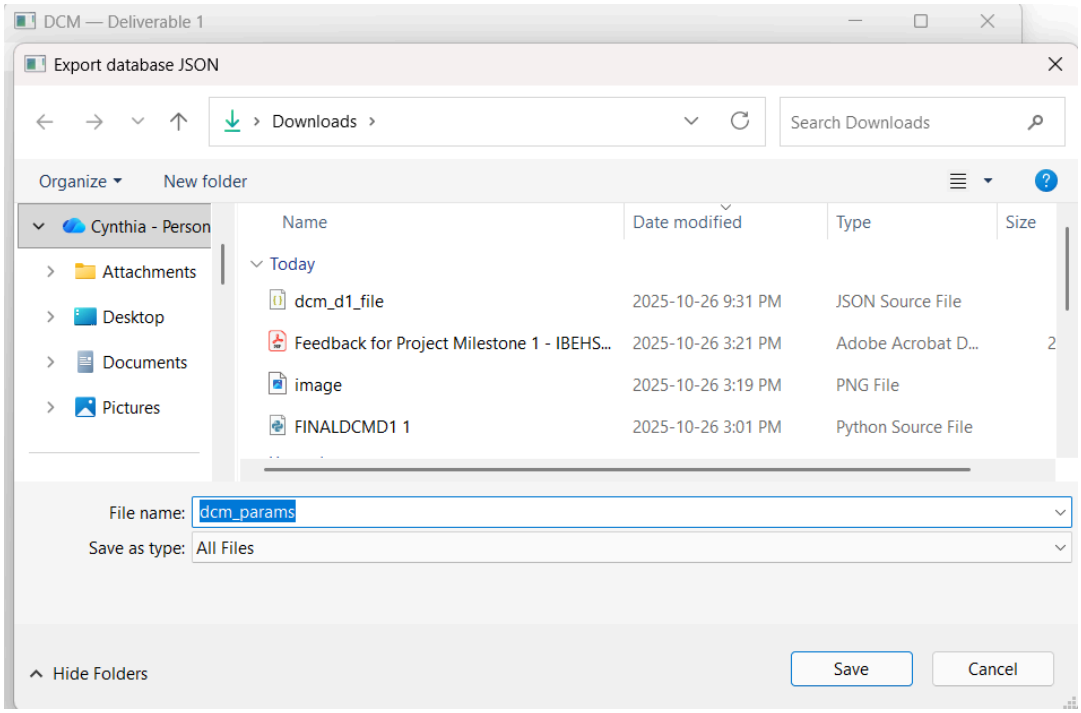

Figure 8: File test with output B

File test with output C (Opens files and allows for the .PDF file “Bradycardia Parameters Report” to be downloaded):

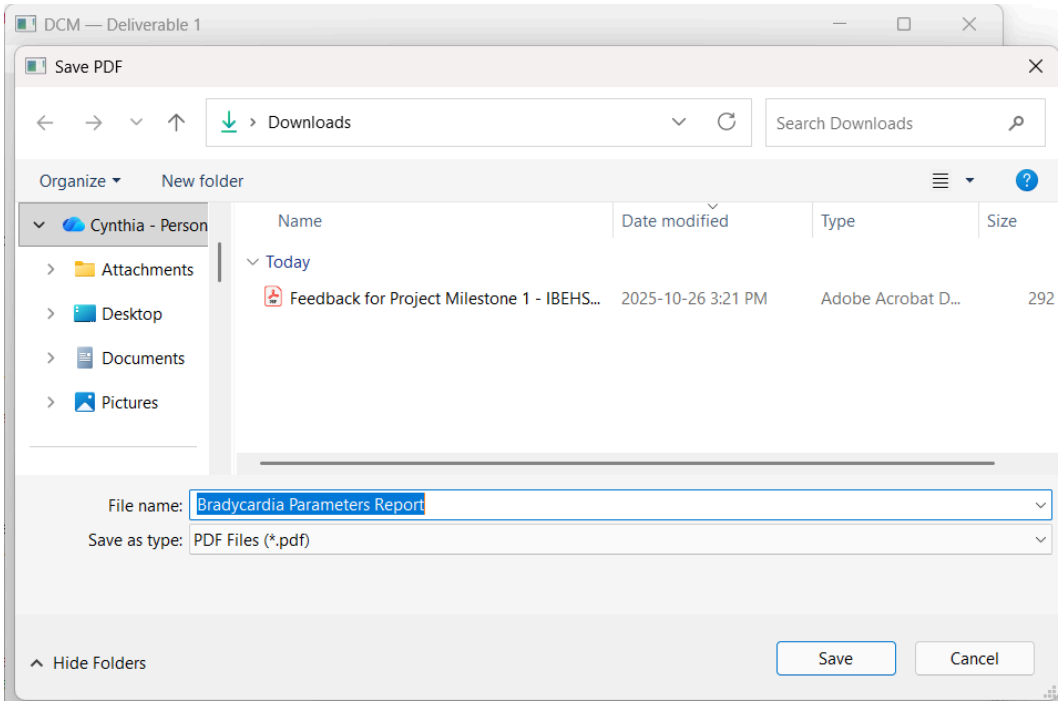

Figure 9: File test with output C opening Save PDF desktop file

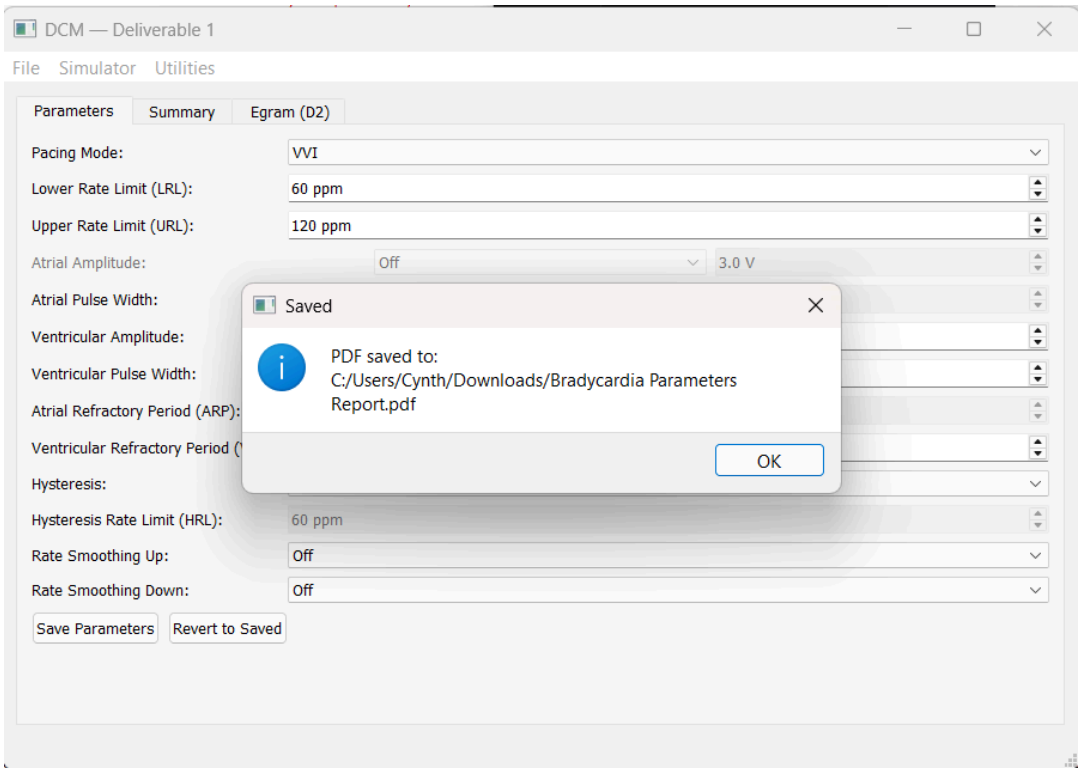

Figure 10: File test with output C with dialog indicating file is saved and the path to it on the computer

Bradycardia Parameters Report

Institution: McMaster University
Printed: 2025-10-27 02:19:21
DCM Model/Version: DCM D1 / D1
DCM Serial: SN
Device ID: None
Report Name: Bradycardia Parameters Report

Mode	VVI
LRL	60 ppm
URL	120 ppm
Atrial Amplitude	Off V
Atrial Pulse Width	0.4 ms
Ventricular Amplitude	3.5 V
Ventricular Pulse Width	0.4 ms
ARP	250 ms
VRP	320 ms
Hysteresis	Off
Rate Smoothing Up	0%
Rate Smoothing Down	0%

Figure 11: File test with output D and the Bradycardia Parameters Report.pdf.

File test with output D (Opens files and allows for the .PDF file “Temporary Parameters Report” to be downloaded):

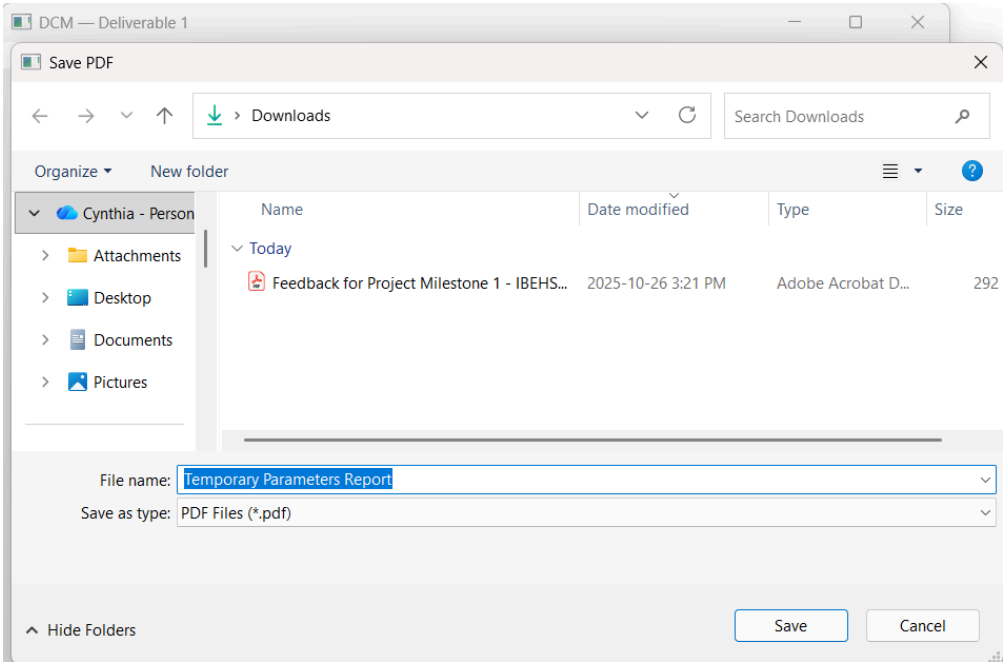

Figure 12: File test with output D and the save for Temporary Parameters Report.pdf on desktop

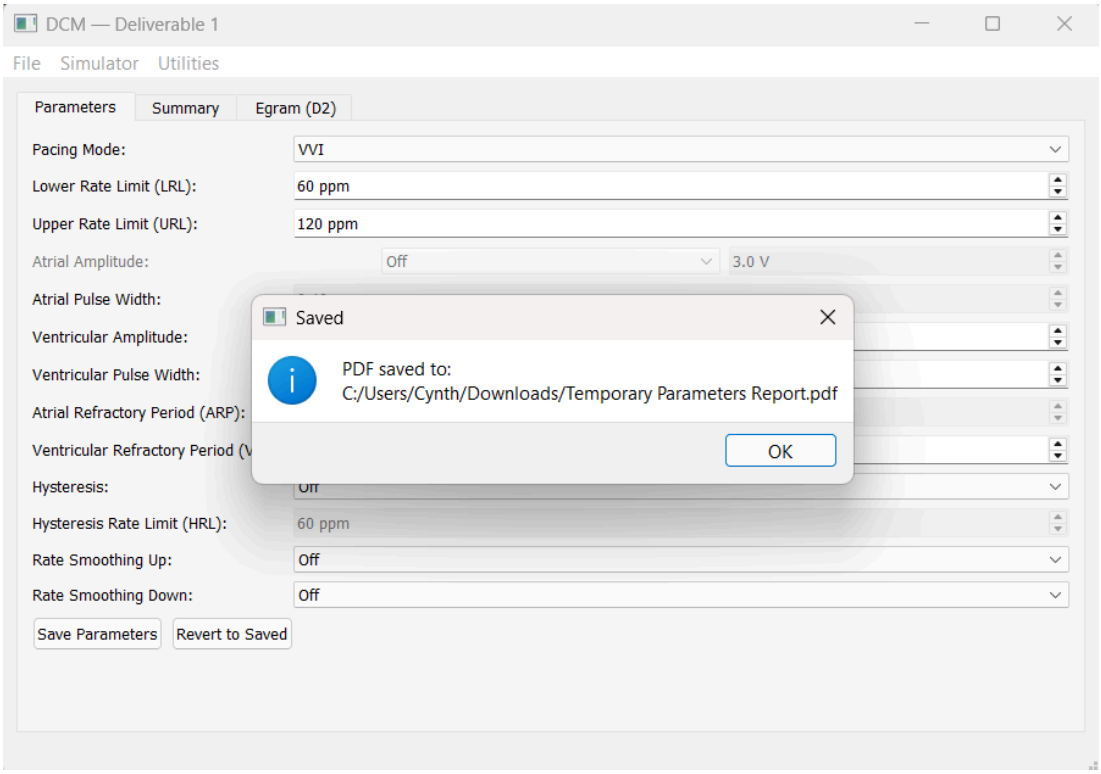

Figure 13: File test with output D with dialog indicating file is saved and its pathway

Temporary Parameters Report

Institution: McMaster University
Printed: 2025-10-27 02:26:19
DCM Model/Version: DCM D1 / D1
DCM Serial: SN
Device ID: None
Report Name: Temporary Parameters Report

Mode	VVI
LRL	60 ppm
URL	120 ppm
Atrial Amplitude	Off V
Atrial Pulse Width	0.4 ms
Ventricular Amplitude	3.5 V
Ventricular Pulse Width	0.4 ms
ARP	250 ms
VRP	320 ms
Hysteresis	Off
Rate Smoothing Up	0%
Rate Smoothing Down	0%

Figure 14: File test with output D and the Temporary Parameters Report.pdf

DCM Simulator dropdown menu test

Simulator test with Toggle Comms Connected, Toggle Device Changed, Set Device ID as default and Telemetry set on Telemetry OK:

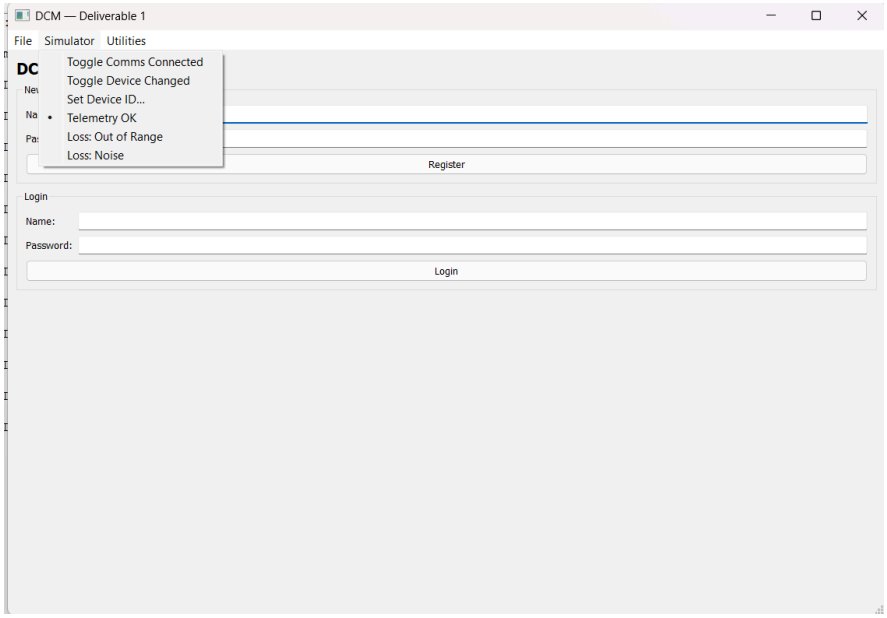

Figure 15: Simulator test showcasing the different togglable tests

Simulator test with outputs B (“Not Connected”), C (“Last Device OK), E (“Device: *None),* G (Telemetry: OK):

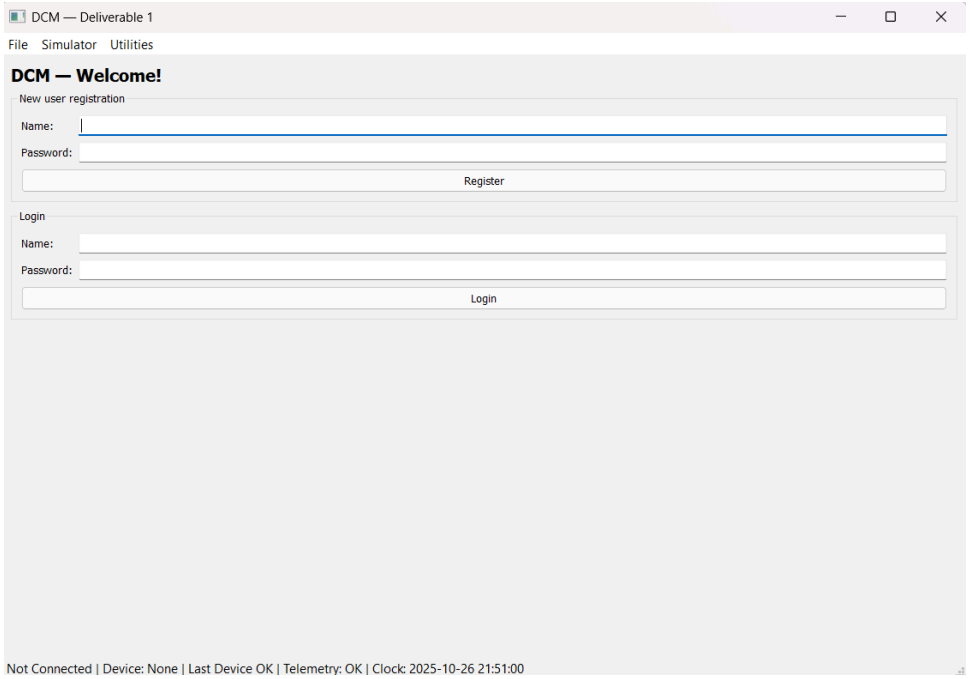

Figure 16: Simulator test outputs B, C and G displayed at the bottom left of the window

Simulator test with outputs A (“Connected”), D (“Device Changed”), F(“Device: <file name>*)*, H (Telemetry: Lost - Out of Range):

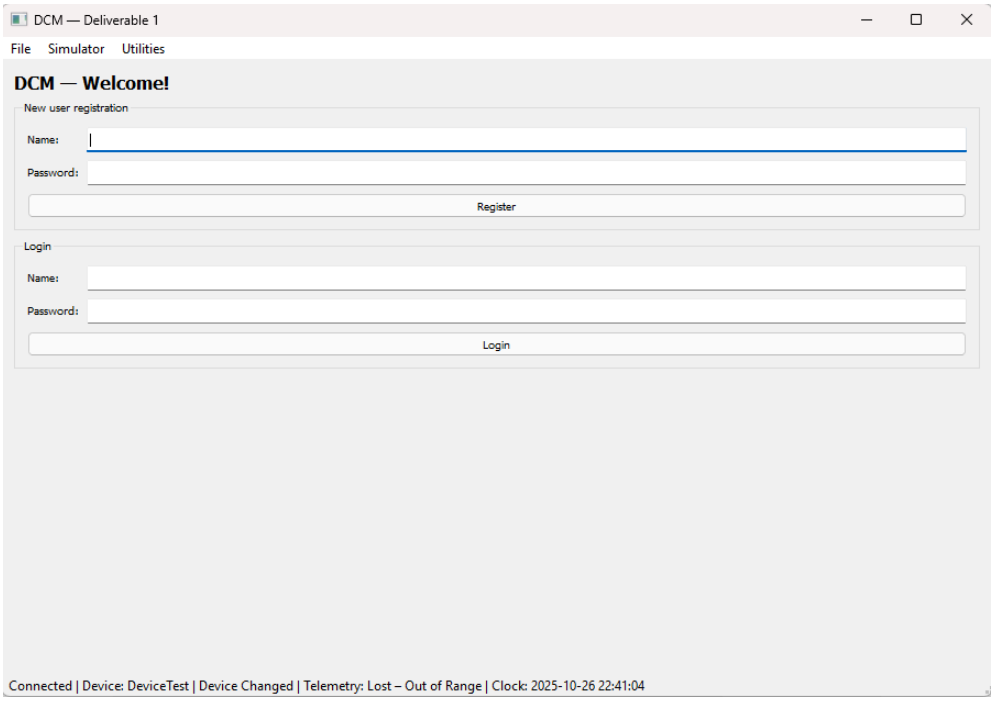

Figure 17: Simulator test outputs A, D, F, and H displayed at the bottom left of the window

Simulator test with outputs B (“Not Connected”), C (“Last Device OK), E (“Device: *None),* I (Telemetry: Lost - Noise):

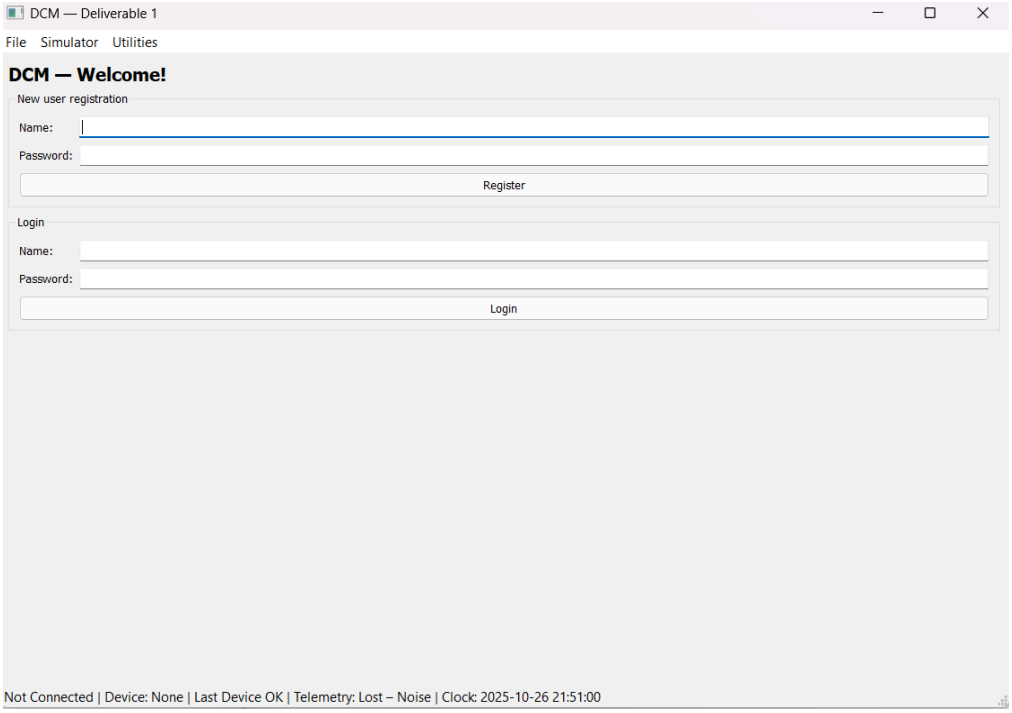

Figure 18: Simulator test outputs B, C, E and I displayed at the bottom left of the window

Simulator test on DashboardPage with outputs J (“Not Connected”), L (“Last Device OK), N (“Device: *None),* P (Telemetry: OK):

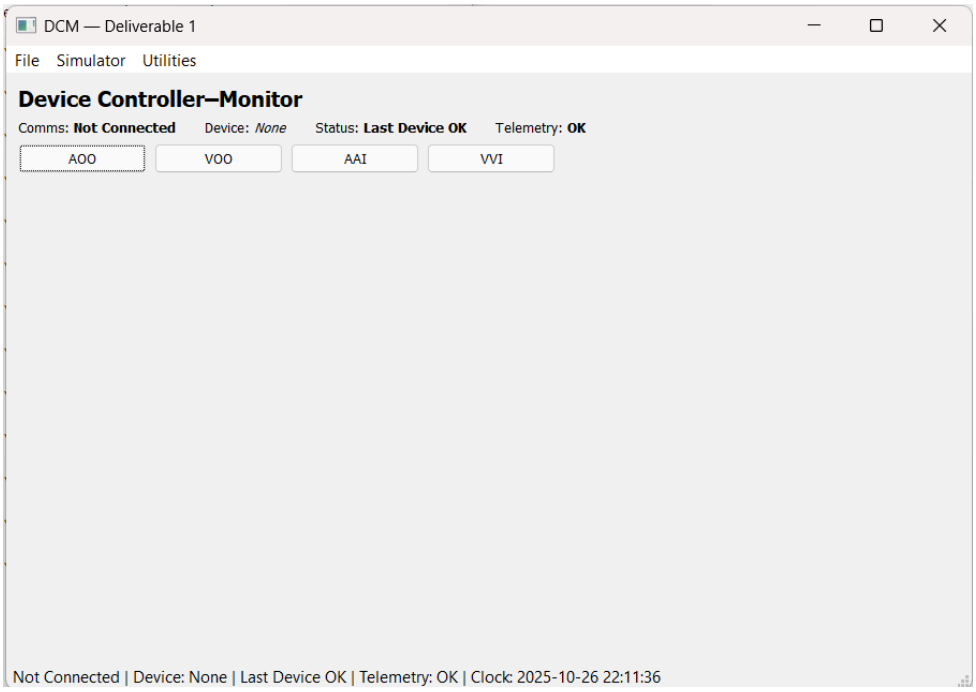

Figure 19: Simulator test outputs J, L, N and P displayed at the bottom left of the window

Simulator test on DashboardPage with outputs K (“Connected”), M (“Device Changed”), O (“Device: <file name>*),* Q (Telemetry: Lost - Out of Range):

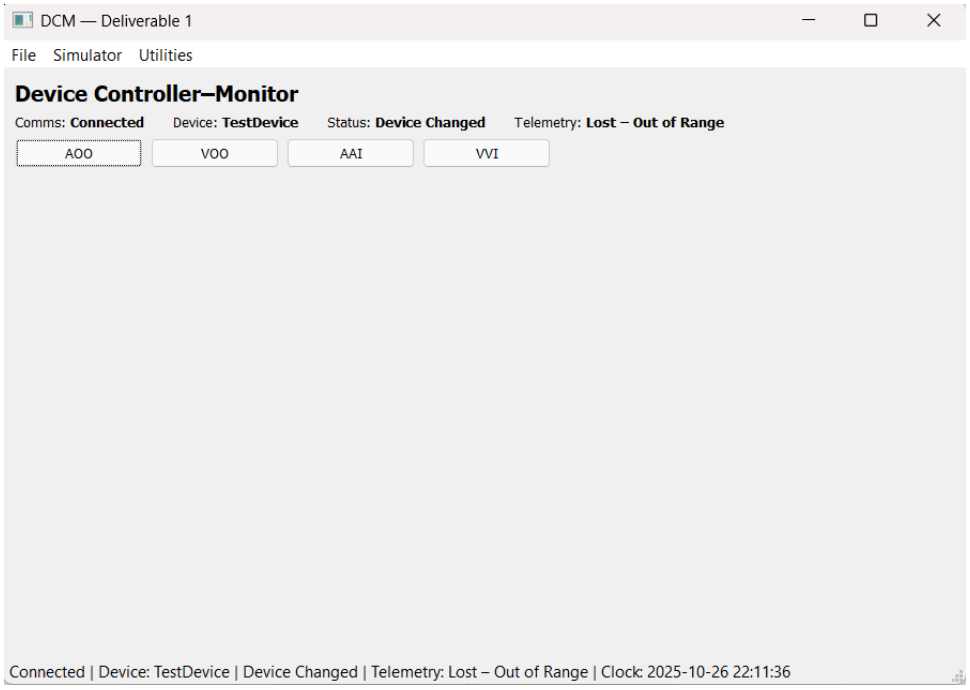

Figure 20: Simulator test outputs K, M, O and Q displayed at the bottom left of the window

Simulator test on DashboardPage with outputs K (“Connected”), M (“Device Changed”), O (“Device: <file name>), R (Telemetry: Lost - Noise):

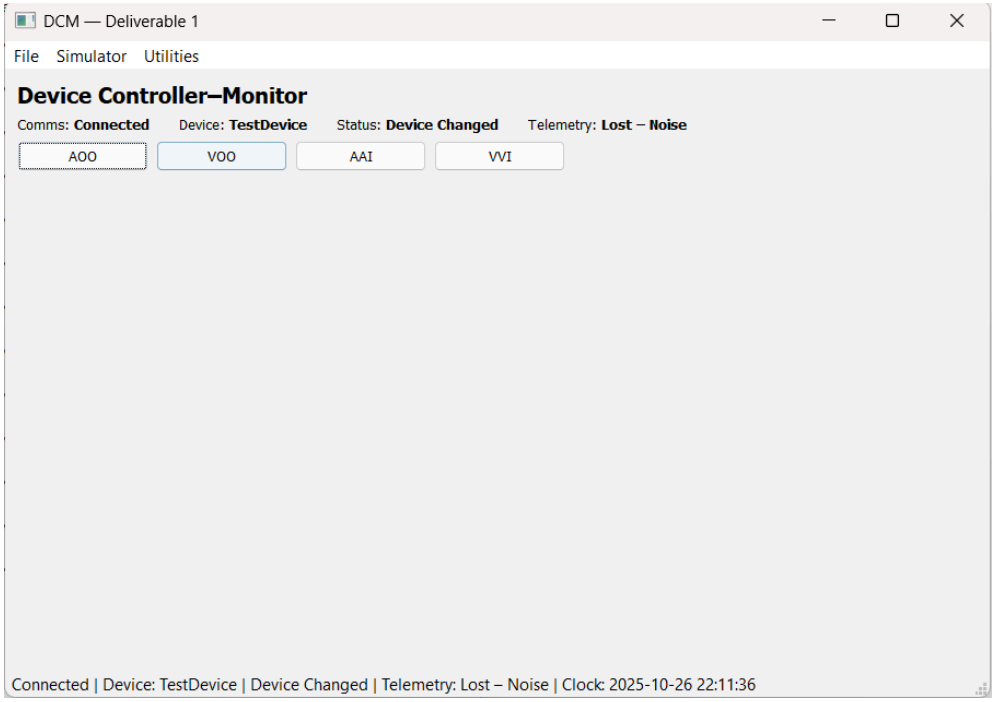

Figure 21: Simulator test outputs K, M, O and R displayed at the bottom left of the window DCM Utilities dropdown menu test

Utilities test with output A (About DCM dialog):

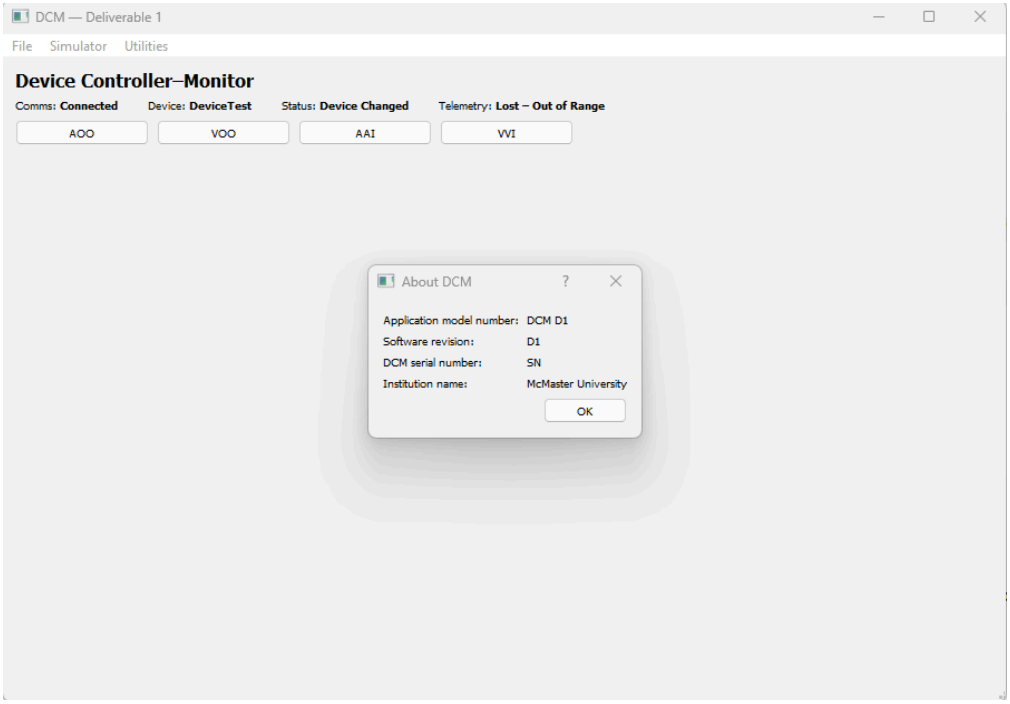

Figure 22: Utilities test output A

Utilities test with output B(The bottom left on each page should display the default current time in the format “yyyy-mm-dd hh:mm:ss”):

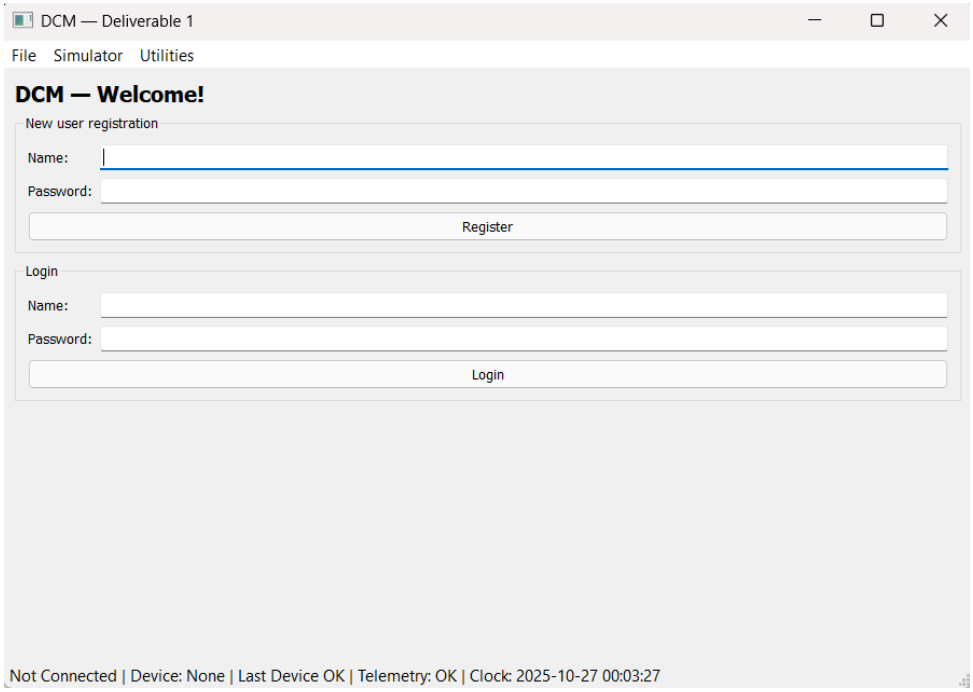

Figure 23: Utilities test output B

Utilities test with output C (Set Clock dialog appears with “Device date/time:” with the format “yyyy-mm-dd hh:mm:ss” editable by typing it):

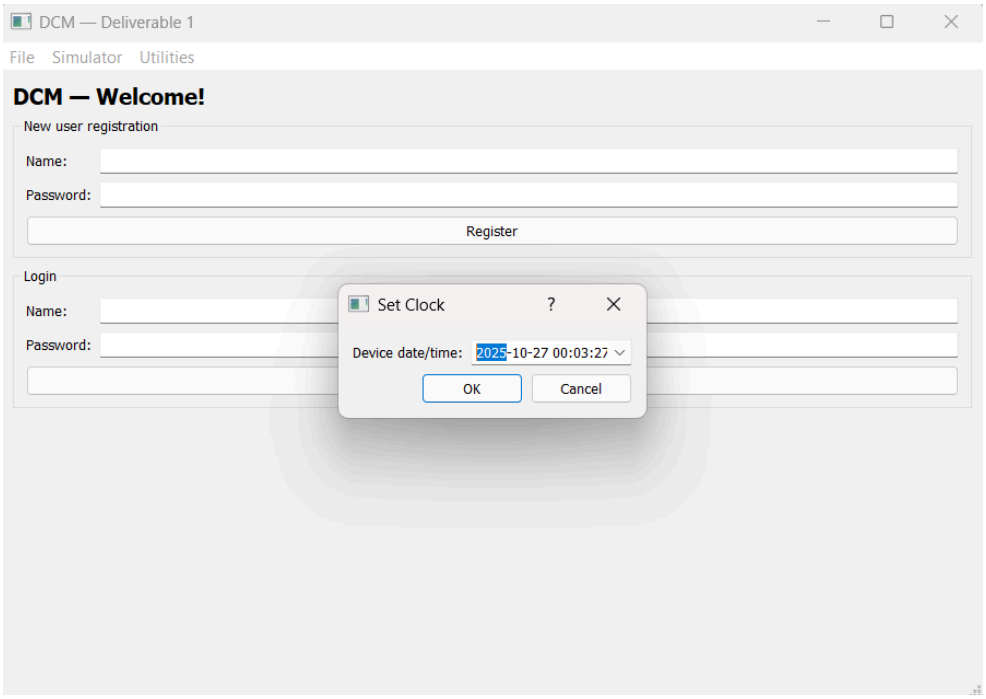

Figure 24: Utilities test output C

Utilities test with output D (dropdown menu for the selectable calendar days displaying the current/saved date in Set Clock):

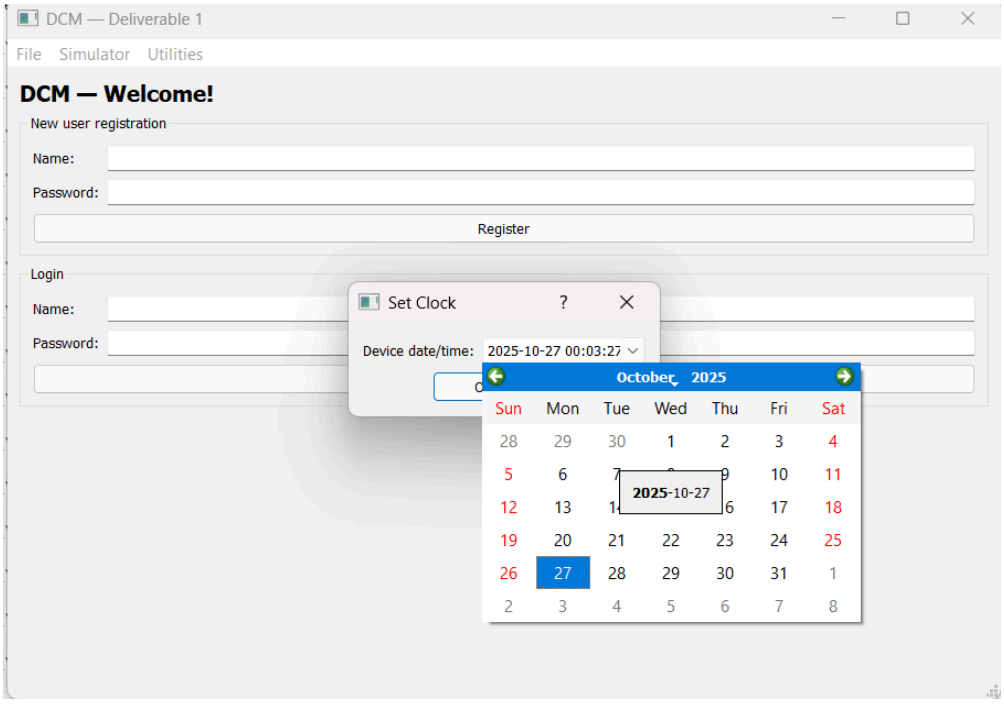

Figure 25: Utilities test output D

Utilities test with output E (The bottom left on each page should display the newly edited time in the format "yyyy-mm-dd hh:mm:ss") :

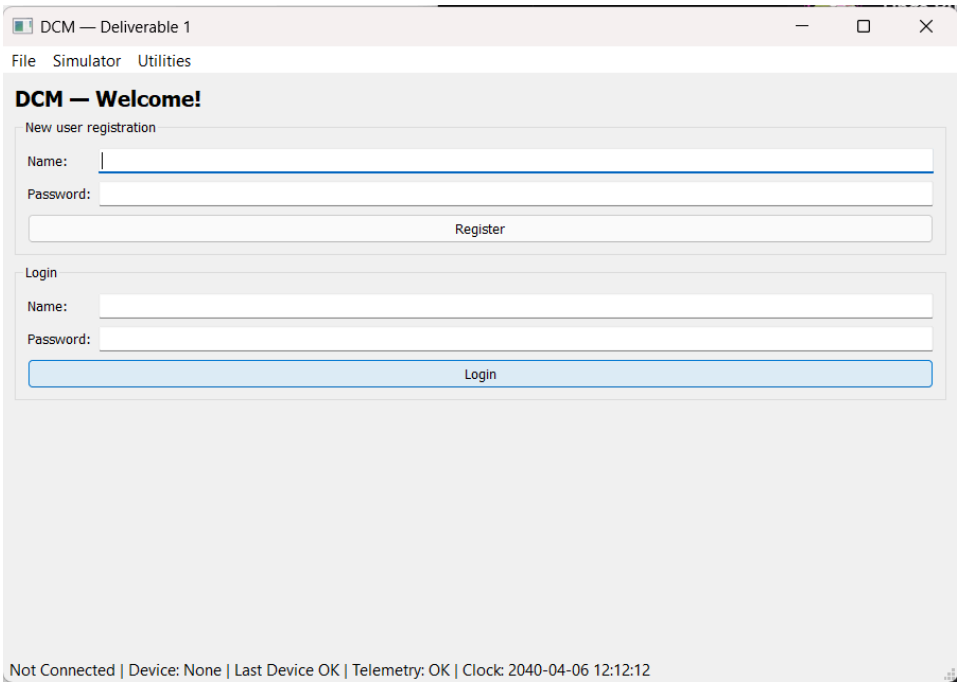

Figure 26: Utilities test output E

DCM Pacing mode and ModeEditorPage tests

Pacing mode and ModeEditorPage test outputs A (AOO parameters), E (default values based on requirements),

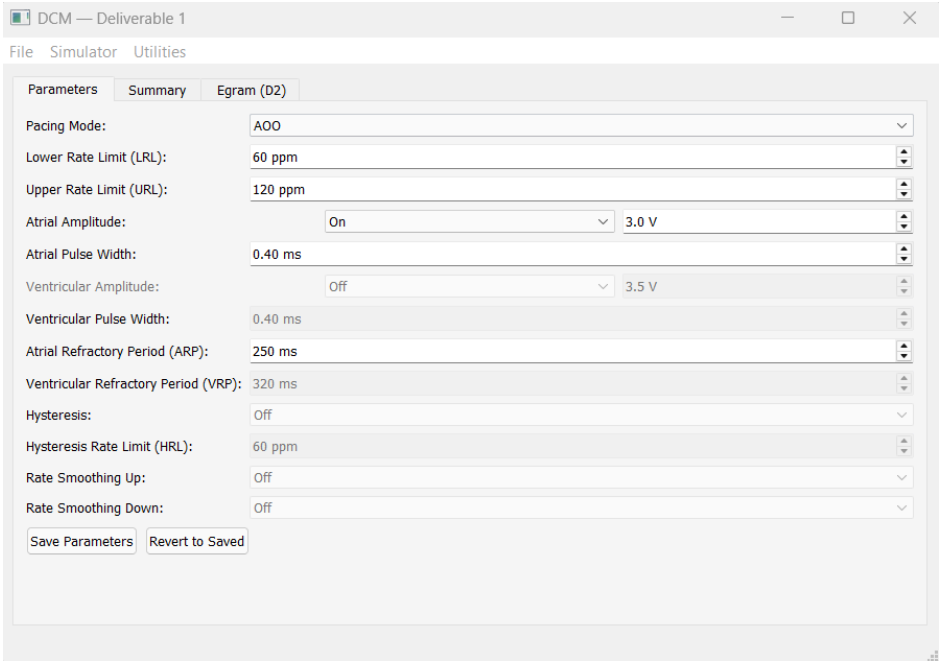

Figure 27: Pacing mode test output A and E

Pacing mode and ModeEditorPage test outputs B (VOO parameters), E (default values based on requirements)

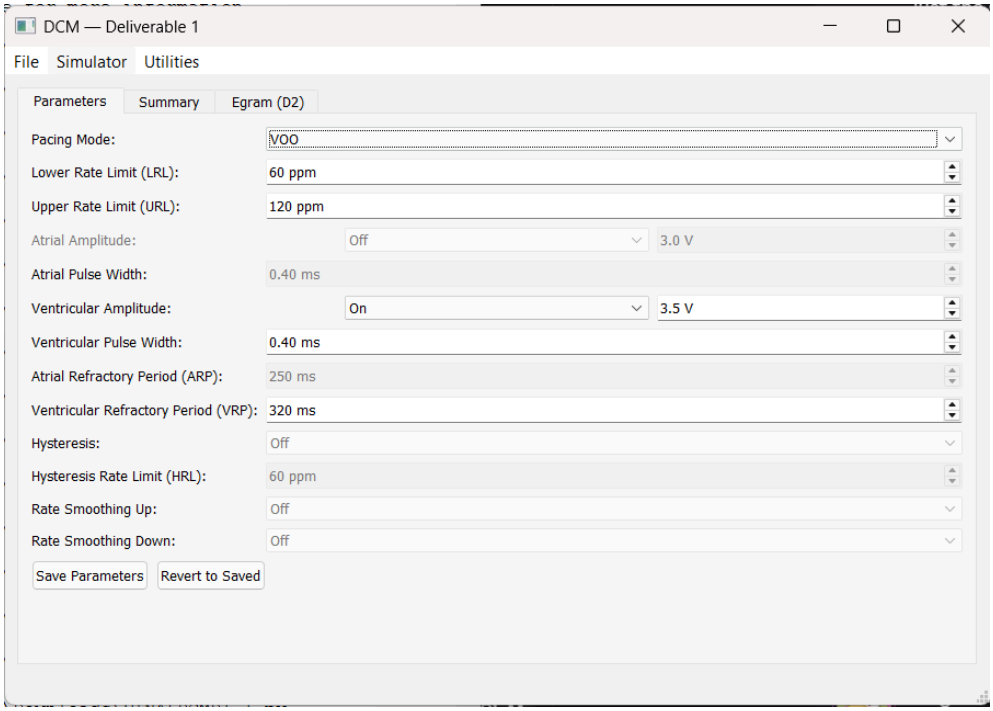

Figure 28: Pacing mode test output B and E

Pacing mode and ModeEditorPage test outputs C (AAI parameters), E (default values based on requirements), output H

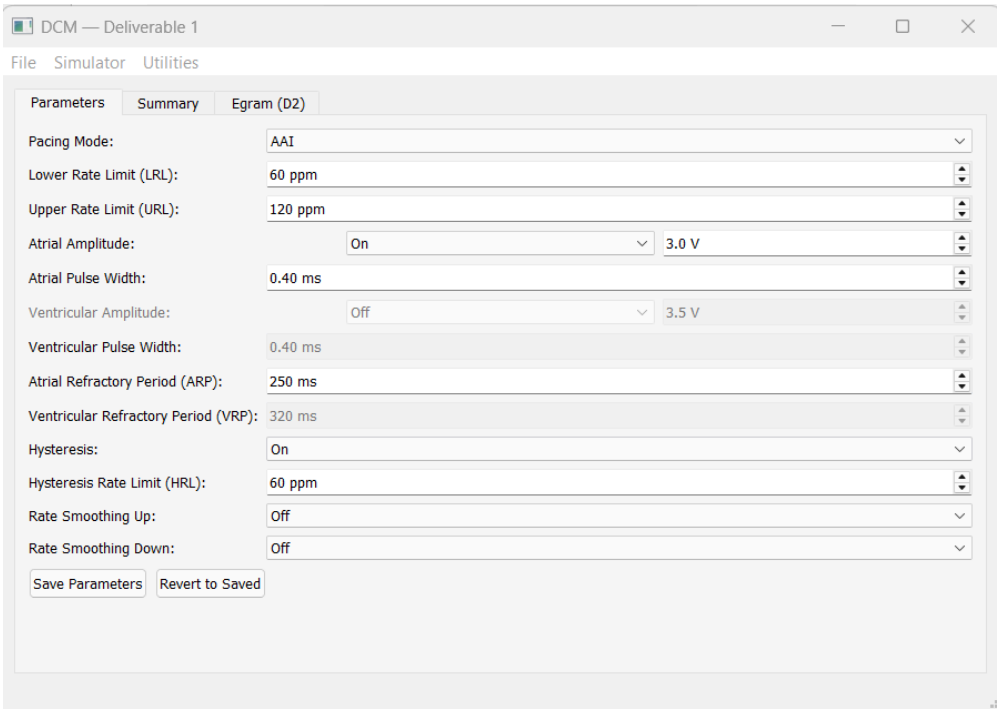

Figure 29: Pacing mode test output C, E and H

Pacing mode and ModeEditorPage test outputs D (VVI parameters), E (default values based on requirements),

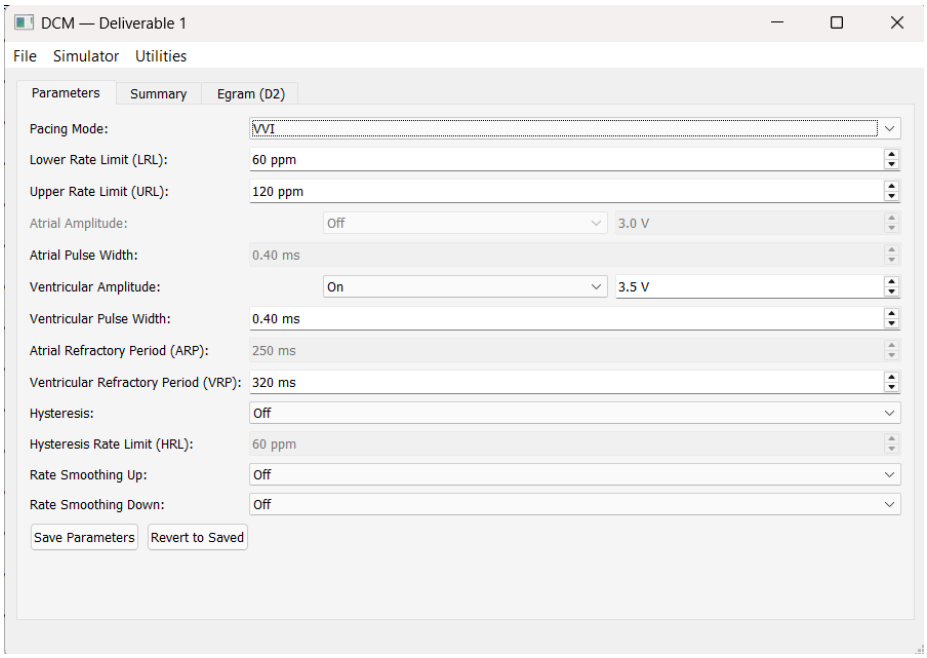

Figure 30: Pacing mode test output D and E

Pacing mode and ModeEditorPage test outputs F (Atrial Amplitude turned off)

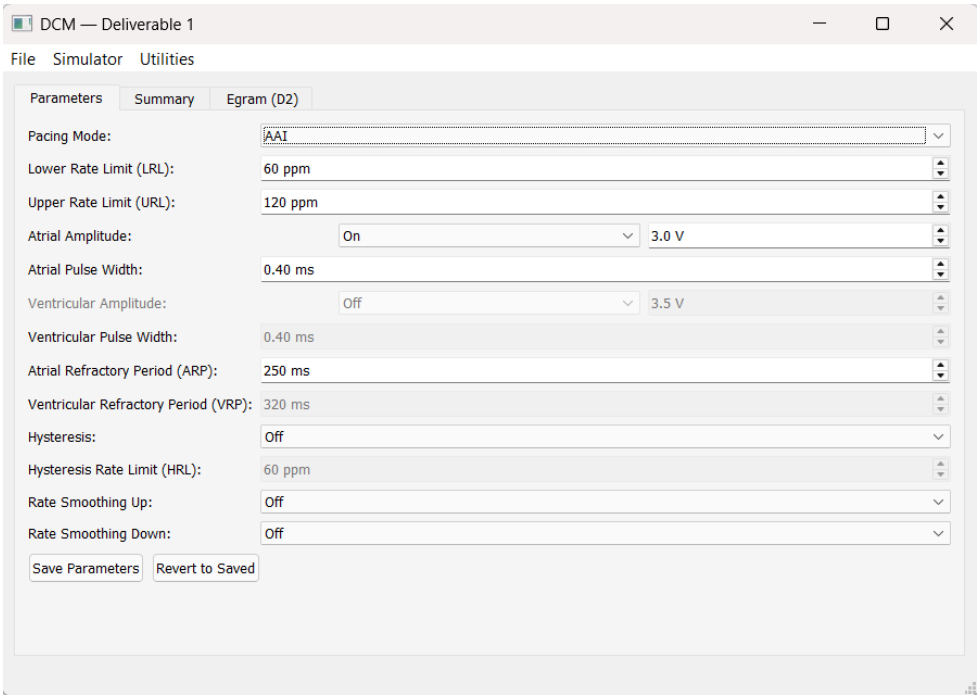

Figure 31: Pacing mode test output F

Pacing mode and ModeEditorPage test outputs G (Ventricular Amplitude turned off), I (HRL not editable).

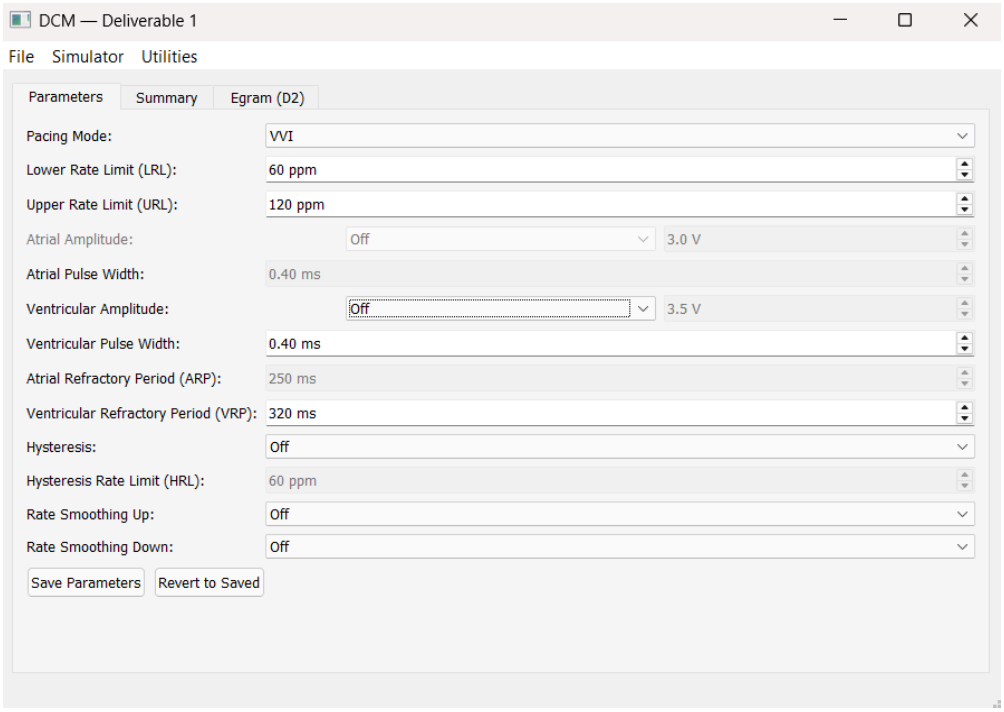

Figure 32: Pacing mode test output G

Pacing mode and ModeEditorPage test outputs J (rate smoothing up dropdown menu)

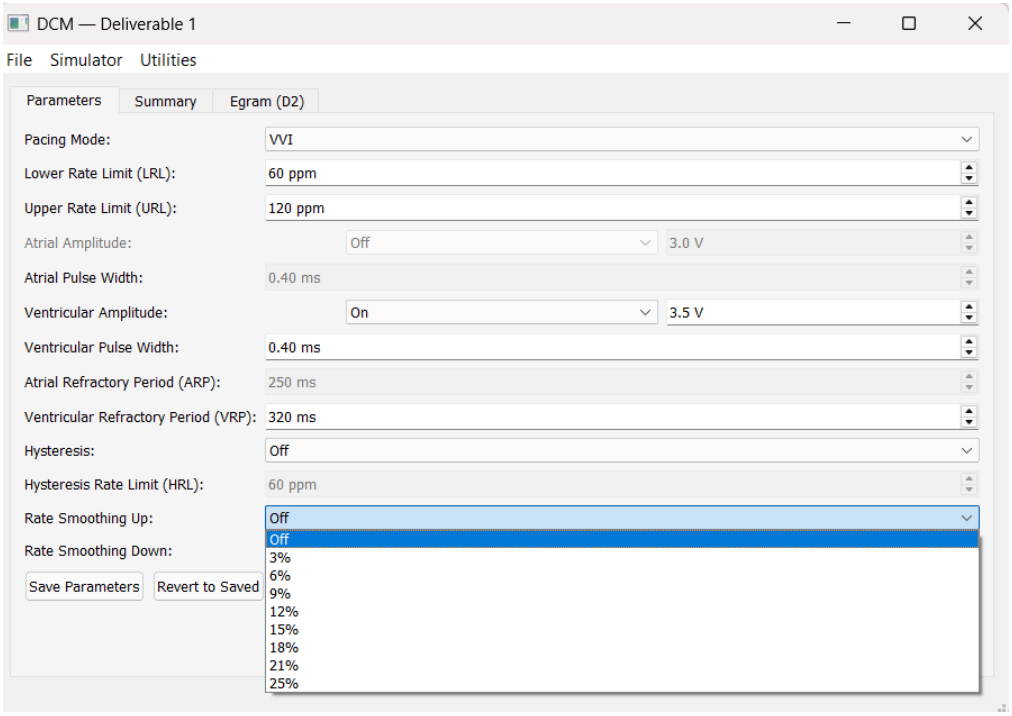

Figure 33: Pacing mode test output J

Pacing mode and ModeEditorPage test outputs K (rate smoothing down dropdown menu)

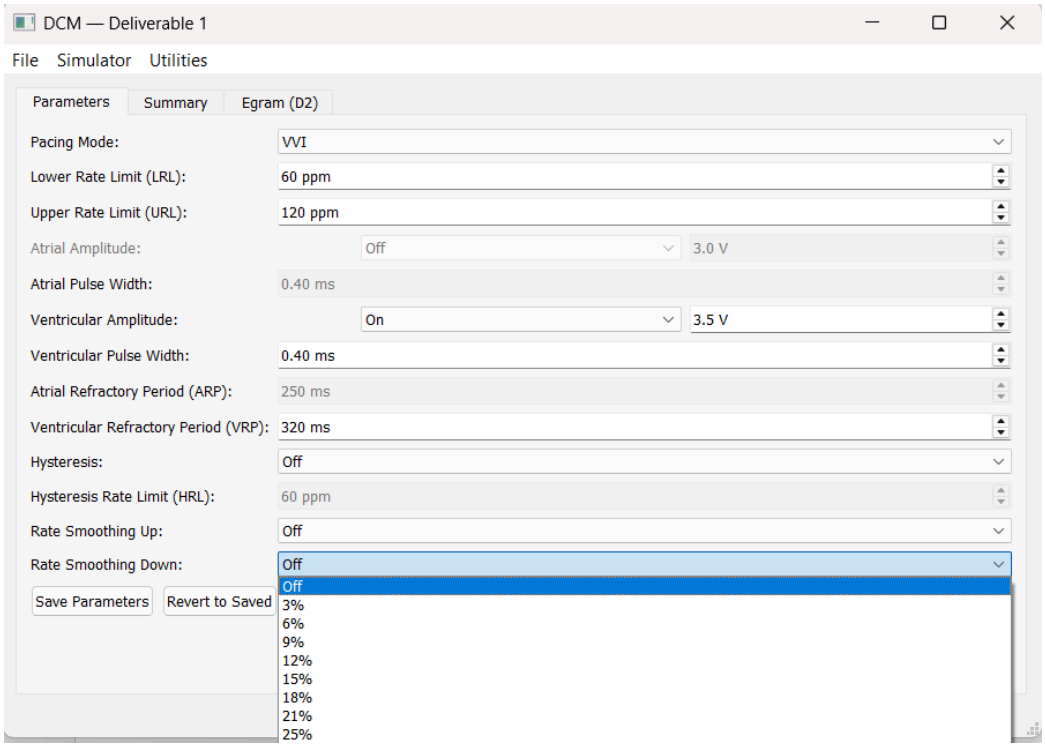

Figure 34: Pacing mode test output K

Pacing mode and ModeEditorPage test outputs L (lowest parameters for all pacing modes)

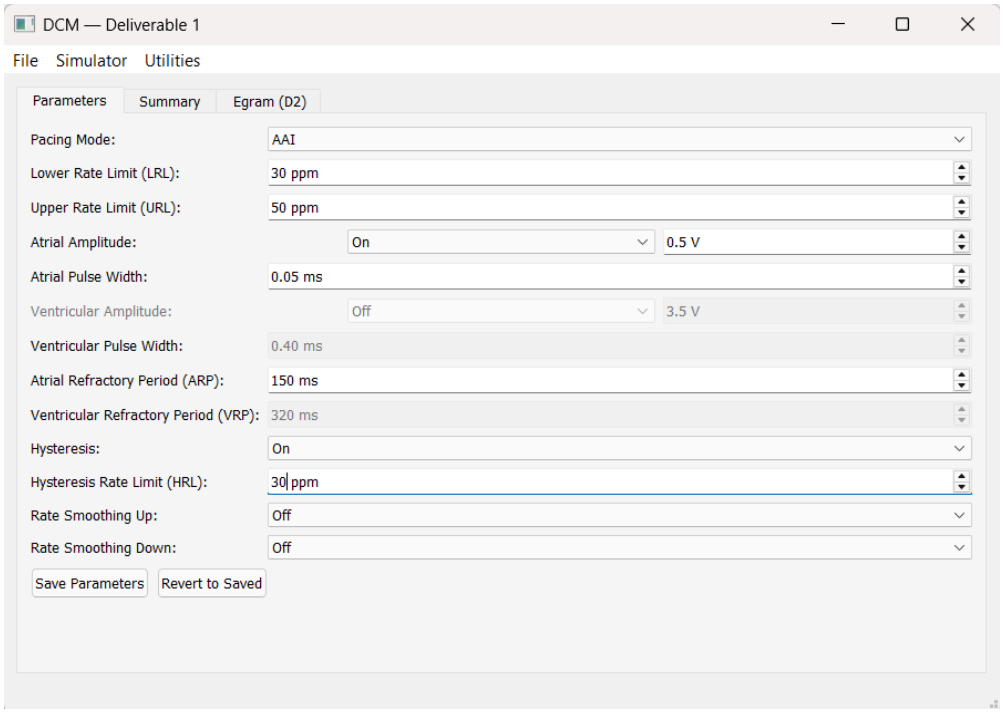

Figure 35: Pacing mode test output L using AAI which will get all the lowest parameters for AOO as well

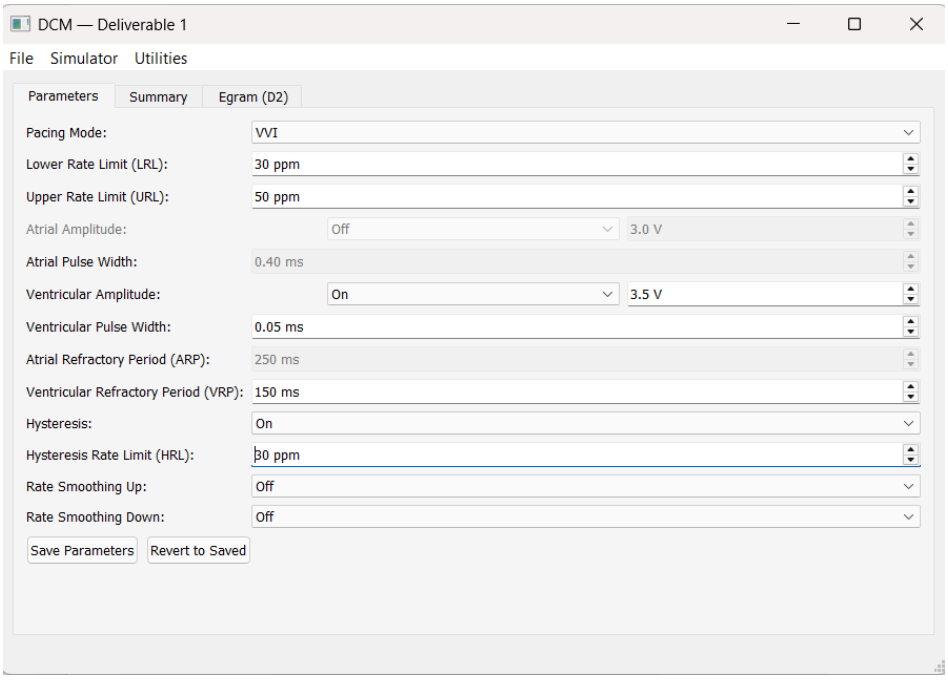

Figure 36: Pacing mode test output L using VVI which will also get the lowest parameters for VOO as well

Pacing mode and ModeEditorPage test outputs M (highest parameters for all pacing modes)

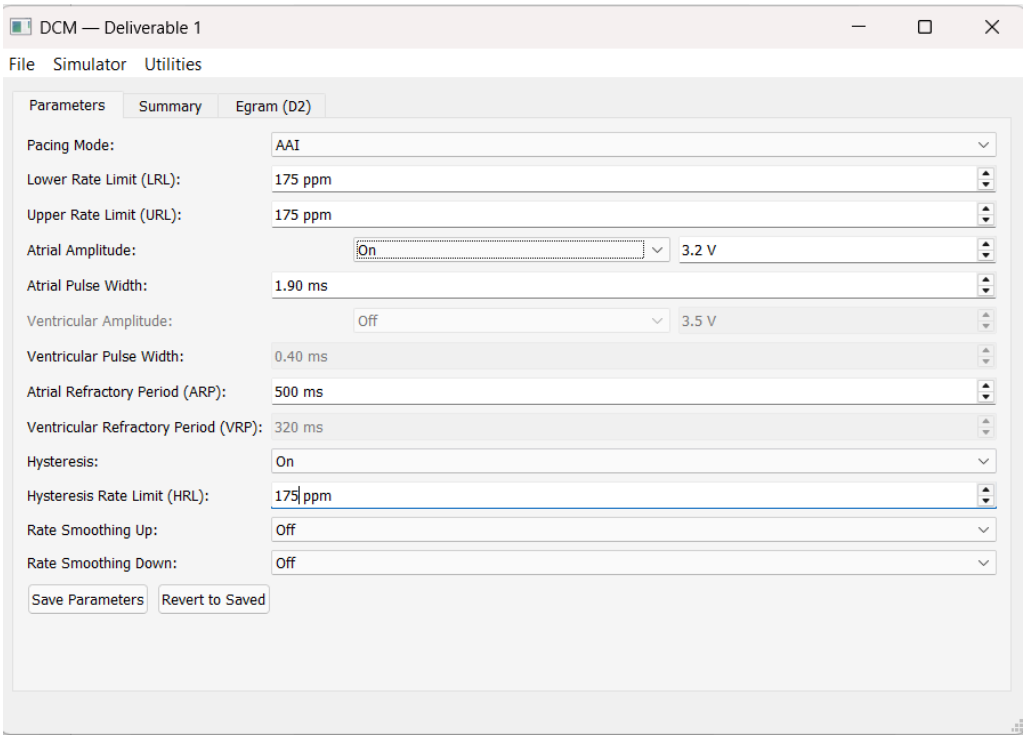

Figure 37: Pacing mode test output M using AAI to get the highest parameters for AOO as well

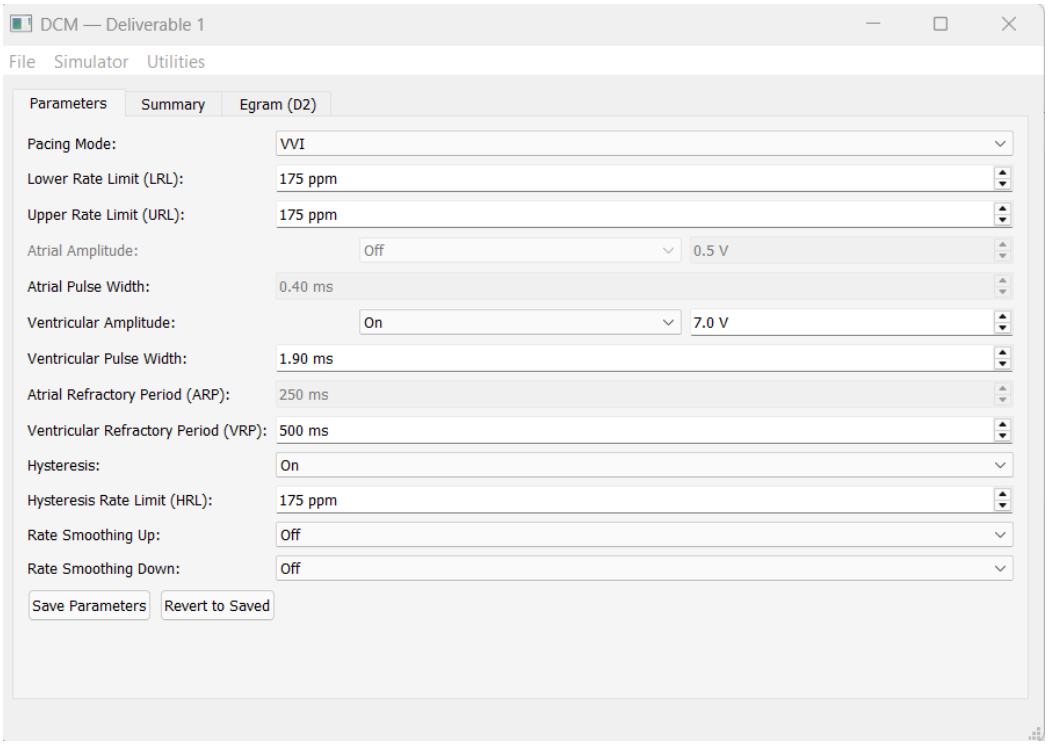

Figure 38: Pacing mode test output M using VVI which will also get the highestparameters for VOO as well

Pacing mode and ModeEditorPage test outputs O (Summary tab)

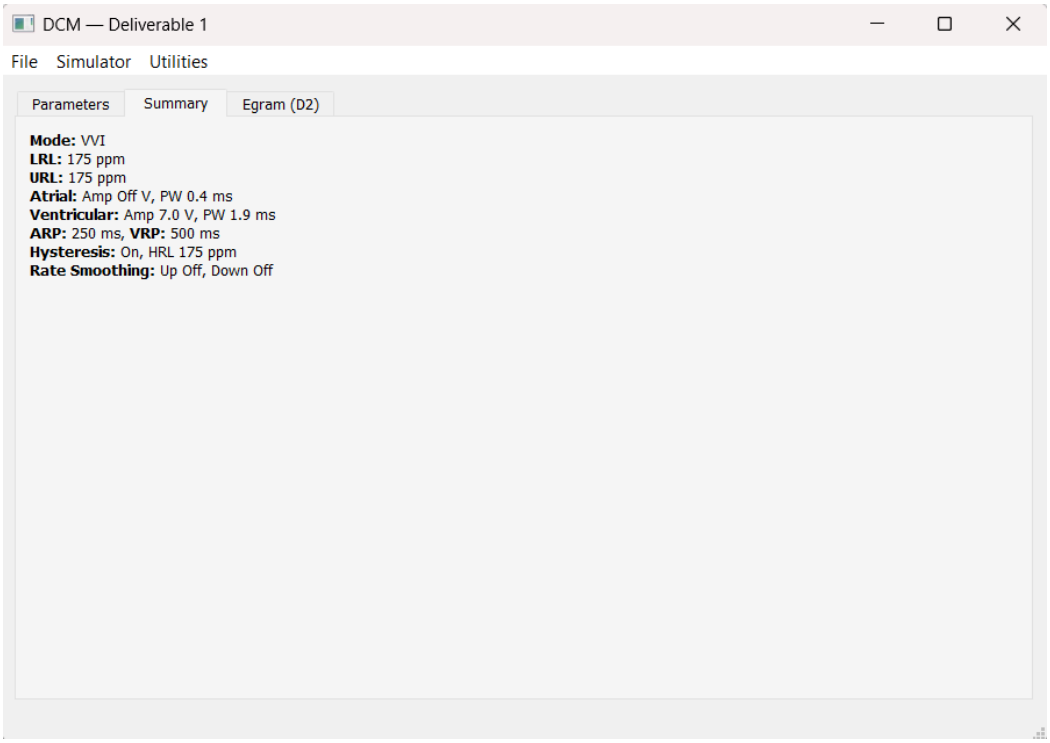

Figure 39: Displays the summary tab page with its corresponding parameters

Pacing mode and ModeEditorPage test outputs P (Egram tab)

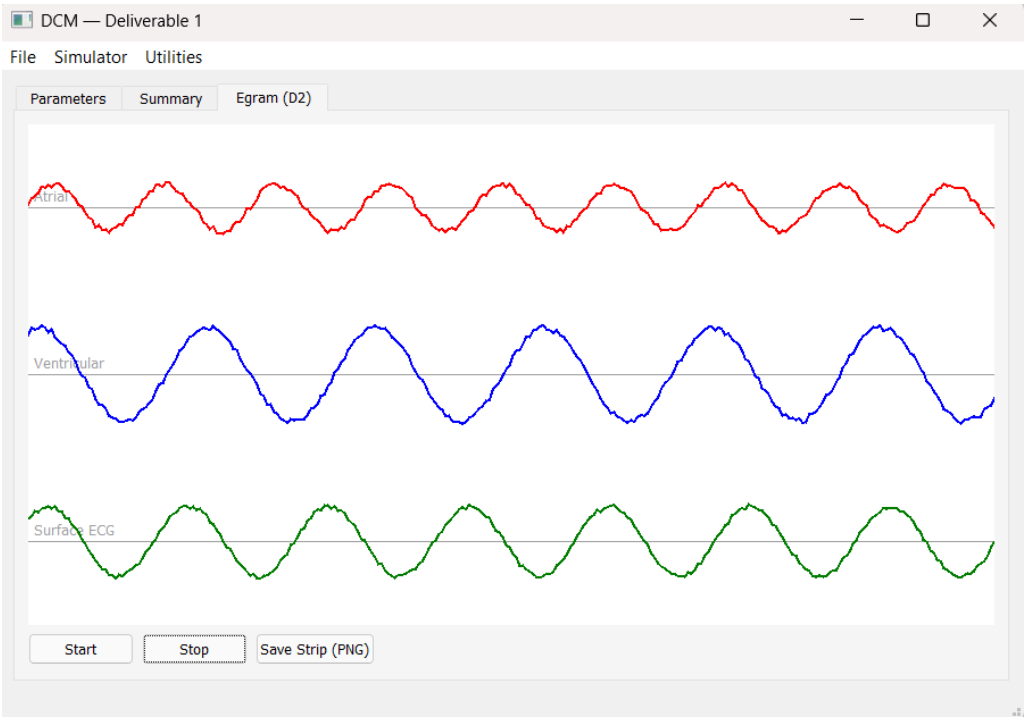

Figure 40: Displays the Egram tab page with the atrial, ventricular, and surface ECG signals.

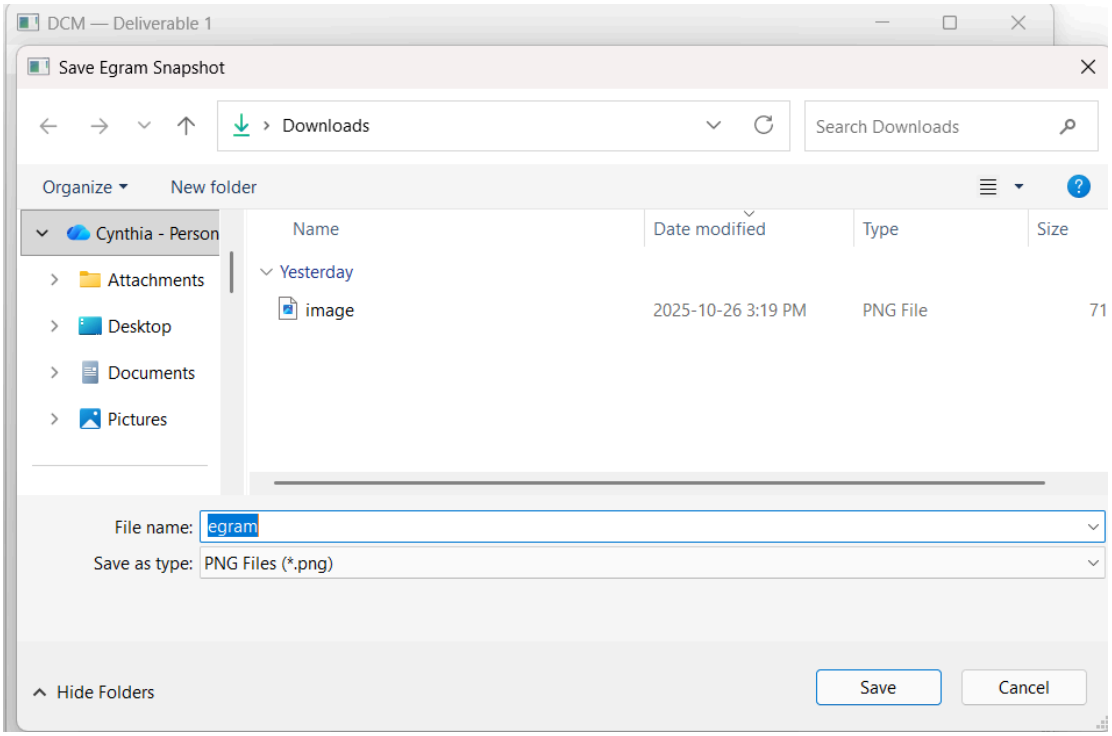

Figure 41: Desktop window opened after pressing Save Strip (PNG) for the Egram file

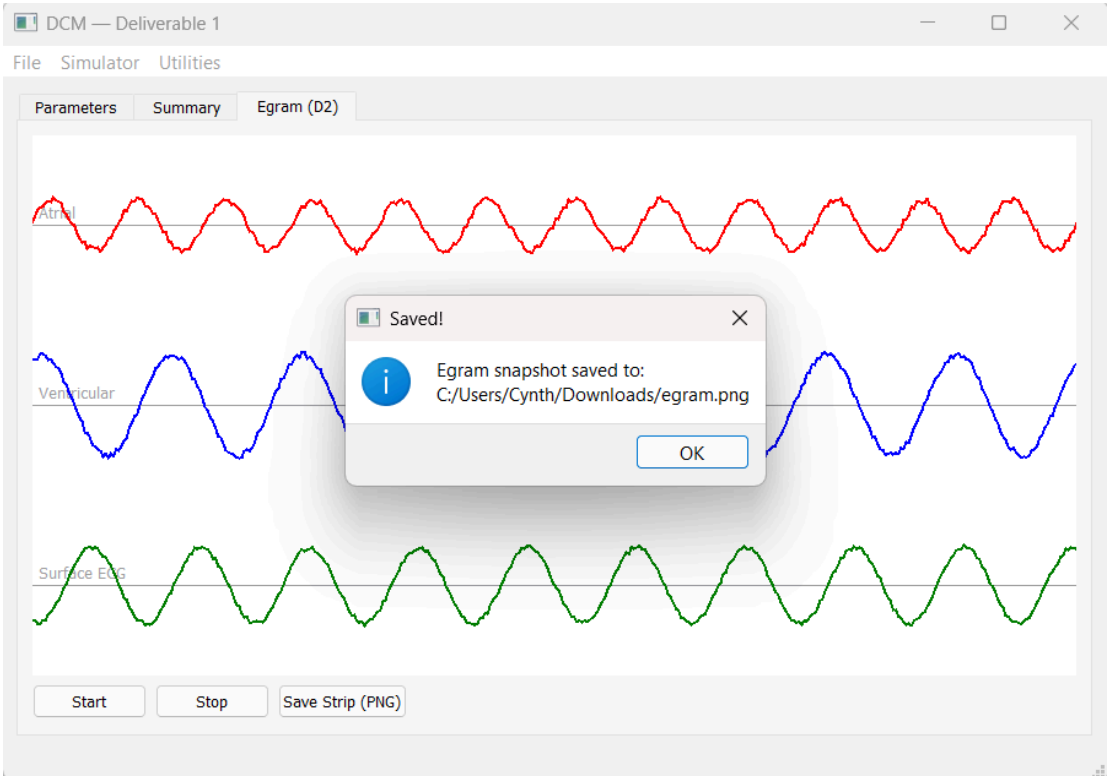

Figure 42: Confirmation of saved strip for Egram file in .png format

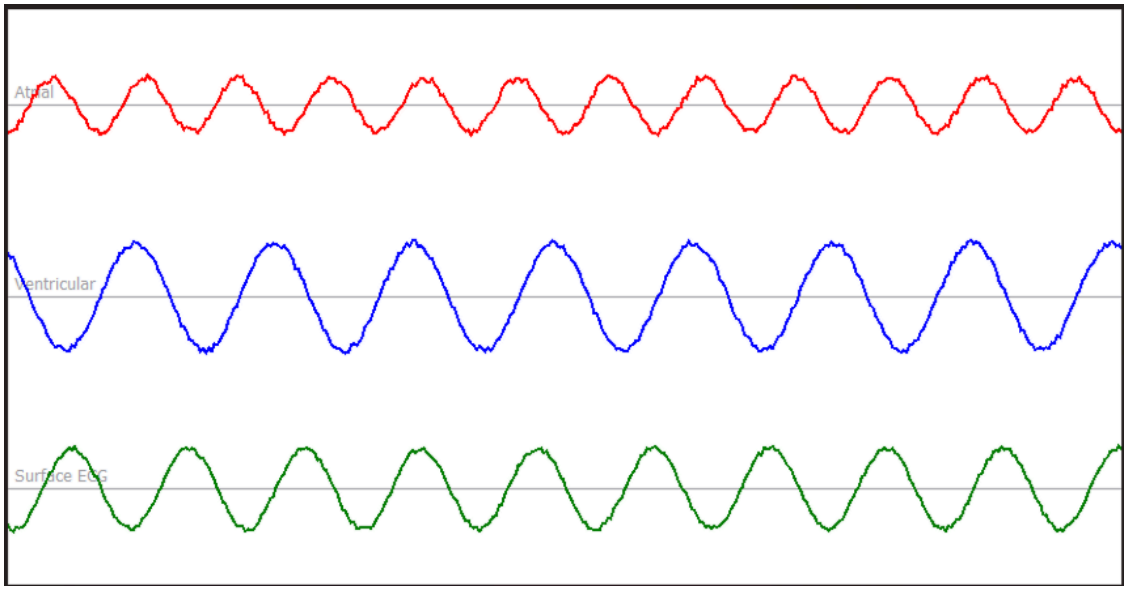

Figure 43: Egram.png

4.3. Appendix 3. DCM Code

```
"""
DCM — Deliverable 1
============================================================

HOW TO RUN
----------
RUN THROUGH IDLE, F5.
NEED PYQT5

MAIN FEATURES
-------------------
- Register/Login with a limit of 10 users (stored locally in JSON; passwords hashed)
- Dashboard  shows comms/device/telemetry state and lets you pick a pacing mode
- Opens a Mode Editor (AOO/VOO/AAI/VVI) with customizable parameters within provided ranges and steps:
    * URL, LRL, A/V Amplitude (+On/Off), A/V Pulse Width, Refractory Period, Hysteresis and Rate Smoothing Up/Down
- Summary tab that mirrors the current parameter set
- Egram (D2) tab with a live simulated 3-trace viewer and option to save PNG
- Exports:
  • File → Reports → Bradycardia Parameters Report…  (PDF)
  • File → Reports → Temporary Parameters Report…    (PDF)
  • File → Export Saved Params JSON…                 (JSON)
- Simulator menu to change states: Comms Connected, Device Changed, Device ID, Telemetry (OK/Out of Range/Noise)
- Utilities → About… and Utilities → Set Clock… (change device time shown in status bar)

DATA IN LOCAL JSON FILE: dcm_d1_file.json
{
  "users":  [ {"username": "<name>", "password_hash": "<sha256>"}, ... ],
  "params": { "<user>:<mode>": { ... parameter dict ... }, ... }
}

Saved parameters are per-(user, mode). Unsaved (current) values are taken directly from the widgets.

WIDGETS emit signals, METHODS connected to them button.clicked.connect(self.on_click),
button.clicked signal

Navigation works by passing CALLBACKS between pages
+ asking MainWindow for global state (active user, simulator flags).
```

```
MAP TO FOLLOW (Objects & Responsibilities)
----------------------------------------
MainWindow (QMainWindow)
  ├─ Database                → JSON read/write for users & params
  ├─ QStackedWidget (self.stack)
  │   ├─ LoginPage           → Register/Login. On success calls MainWindow._on_login_ok(name)
  │   ├─ DashboardPage       → Shows status labels + 4 mode buttons; calls
MainWindow._open_mode_editor(mode)
  │   └─ ModeEditorPage      → Parameter widgets, Summary, Egram
  │       ├─ Uses Database   → save_params()/load_params() for (user, mode)
  │       └─ D1EgramView     → Live simulated traces; Start/Stop/Save PNG
  ├─ Menus
  │   ├─ File → Reports      → _export_brady_params_report(), _export_temp_params_report()
  │   ├─ File → Export JSON  → _export_all_json()
  │   ├─ Simulator           → _toggle_comms(), _toggle_device_changed(),
  │   │                      _set_device_id(), _set_telemetry("ok|out_of_range|noise")
  │   └─ Utilities           → _show_about(), _set_clock_dialog()
  └─ QStatusBar              → Shows Comms | Device ID | Change state | Telemetry | Clock

"""

# ==========================
# 1) Standard library imports
# ==========================
import json # read/write JSON file (database)
import os # file paths and existence checks
import hashlib # SHA256 to hash passwords (security)
from typing import Dict, Any, List, Optional  # type hints

# ==========================
# 2) PyQt5 GUI imports
# ==========================
from PyQt5.QtCore import Qt, QSize, QDateTime, QTimer # QtCore has core types; Qt namespace
from PyQt5.QtGui import QIcon, QValidator, QTextDocument # QtGui has visual stuff (icons, validators)
from PyQt5.QtPrintSupport import QPrinter # printsupport for pdfs, 3.2.4


from PyQt5.QtGui import QIcon, QValidator

from PyQt5.QtWidgets import (
    QApplication, # the global GUI application object (event loop here)
    QWidget, # base class for most things shown on screen
    QMainWindow, # main window with menu/status/central widget
    QStackedWidget, # "deck" to switch between multiple pages
```

```
    QVBoxLayout, # vertical layout manager
    QHBoxLayout, # horizontal layout manager
    QLabel, # shows text or rich text (HTML)
    QLineEdit, # single-line text input
    QPushButton, # clickable button
    QMessageBox, # modal dialogs (info/warning/errors)
    QFormLayout, # 2-column "Label : Field" layout
    QSpinBox, # integer input with arrows + typing
    QDoubleSpinBox, # floating-point input with arrows + typing
    QComboBox, # drop-down selection list
    QGroupBox, # box with a title around a group of widgets
    QTabWidget, # tabs (parameters / summary / egram)
    QFileDialog, # file picker dialog
    QAction, # menu item (can be clickable, checkable)
    QActionGroup, # group of actions (useful for radio-button behavior)
    QStatusBar,  # one-line bar at bottom for status text
    QDialog, # popup
    QDialogButtonBox, #buttons
    QDateTimeEdit #change time 3.2.3
)

# ============================
# 3) FILE STORAGE!!!!!!!!!!
# ============================
DB_FILE = "dcm_d1_file.json" # JSON file path to store users + saved parameters
MAX_USERS = 10 # allow at most 10 users locally

class Database:
    """
    We store JSON like:
    {
      "users": [{"username": "alice", "password_hash": "..."}, ...],
      "params": {"alice:VVI": {...}, "alice:AAI": {...}, ...}
    }
    """

    def __init__(self, path: str): #string, path for location of json file, will be in current working directory
        self.path = path # saving incoming string path on the object itself as self.path so it can be used later

        # true if file exists, if not flips it
        if not os.path.exists(self.path):  # if file doesn't exist yet
            self._write({"users": [], "params": {}}) # create with empty structure

    # private read helper _ internal
```

```
def _read(self) -> Dict[str, Any]:
    with open(self.path, "r", encoding="utf-8") as f: # open file for reading
        return json.load(f)  # parse JSON -> Python dict

# private write helper
def _write(self, data: Dict[str, Any]) -> None:
    with open(self.path, "w", encoding="utf-8") as f: # open for writing (overwrite)
        json.dump(data, f, indent=2) # dump dict -> json

# public: count how many users are registered
def user_count(self) -> int:
    return len(self._read()["users"])

# public: add a new user (case-insensitive unique usernames; enforce MAX_USERS)
def add_user(self, username: str, password_hash: str) -> bool:
    data = self._read()
    # check duplicate ignoring case (use .lower() on both sides)
    if any(u["username"].lower() == username.lower() for u in data["users"]):
        return False
    # enforce maximum user count
    if len(data["users"]) >= MAX_USERS:
        return False
    # append new record and save to file
    data["users"].append({"username": username, "password_hash": password_hash})
    self._write(data)
    return True

# public: verify login (username+hash must match a stored record)
def verify_user(self, username: str, password_hash: str) -> bool:
    data = self._read()
    user_lc = username.lower()
    for user in data["users"]:
        if user["username"].lower() == user_lc and user["password_hash"] == password_hash:
            return True
    return False

# public: save parameter dict under key "<user>:<mode>"
def save_params(self, username: str, mode: str, params: Dict[str, Any]) -> None:
    data = self._read()
    key = f"{username}:{mode}" # composite key simple lookup
    data["params"][key] = params
    self._write(data)

# public: load parameter dict; if none saved, return empty dict
```

```
    def load_params(self, username: str, mode: str) -> Dict[str, Any]:
        data = self._read()
        return data["params"].get(f"{username}:{mode}", {})

# ==========================
# 4) Parameters
# ==========================

# LRL (Lower Rate Limit) has piecewise steps depending on the range
LRL_SEGMENTS = [
    (30, 50, 5),   # 30, 35, 40, 45, 50
    (50, 90, 1),   # 50..90 (1 step)
    (90, 175, 5),  # 90, 95, ..., 175
]
# URL (Upper Rate Limit): 50–175 step 5
URL_MIN, URL_MAX, URL_STEP = 50, 175, 5

# Amplitude choices: "Off", 0.5–3.2 (0.1 step) and 3.5–7.0 (0.5 step)
AMP_LOW_MIN,  AMP_LOW_MAX,  AMP_LOW_STEP  = 0.5, 3.2, 0.1 # SHOULD BE FOR
ATRIAL
AMP_HIGH_MIN, AMP_HIGH_MAX, AMP_HIGH_STEP = 3.5, 7.0, 0.5 # SHOULD BE VENT

# Pulse width choices: 0.05 ms single value, then 0.1–1.9 in 0.1 steps
PW_SEGMENTS = [
    (0.05, 0.1, 0.05),  # only 0.05
    (0.1, 1.9, 0.1),
]

# Refractory periods: 150–500 ms in 10 ms steps
REF_MIN, REF_MAX, REF_STEP = 150, 500, 10

# D1 modes (used to generate buttons and choices)
MODES = ["AOO", "VOO", "AAI", "VVI"]

# Hysteresis:
# - enabled only for inhibiting modes (AAI, VVI)
# - when enabled, the Hysteresis Rate Limit (HRL) uses the same choices as LRL
HYSTERESIS_STATES = ["Off", "On"]

# Rate Smoothing:
# - two programmable parameters Up and Down
# - options Off, 3, 6, 9, 12, 15, 18, 21, 25 %
RATE_SMOOTH_CHOICES = ["Off", "3%", "6%", "9%", "12%", "15%", "18%", "21%", "25%"]
```

```
# -------------------------
# About / Utility constants
# -------------------------
APP_MODEL_NUMBER   = "DCM D1"
APP_SOFTWARE_REV   = "D1"
APP_SERIAL_NUMBER  = "SN"
APP_INSTITUTION    = "McMaster University"

# ============================
# 5) helpers
# ============================
def hash_password(plain: str) -> str: # hash password safety
    """
    Turn a plaintext password into a SHA256 hex string.
    - .encode("utf-8") converts Python str to bytes
    - hashlib.sha256(...).hexdigest() returns a hex string like 'a9f...'
    """
    return hashlib.sha256(plain.encode("utf-8")).hexdigest()

def build_allowed_lrl(segments) -> List[int]: # doesnt need float range bc whole numbers
    """
    build allowed integer LRL values by merging each (lo,hi,step) segment
    """
    vals = set() # set prevents duplicates around edges
    for lo, hi, st in segments:
        vals.update(range(lo, hi + 1, st))  # +1 makes hi inclusive
    return sorted(vals)


def build_allowed_url(min_v: int, max_v: int, step: int) -> List[int]: # doesnt need float range bc whole
numbers
    """simple inclusive range for URL values"""
    return list(range(min_v, max_v + 1, step))

def build_pw_values() -> List[float]:
    """
    build pulse-width allowed values from the PW_SEGMENTS spec.
    uses set to avoid duplicates; sorted to make stepping nice.
    """
    vals = set()
    for lo, hi, st in PW_SEGMENTS:
        x = lo
        while x <= hi + 1e-12: # include hi
            vals.add(round(x, 2)) # keep two decimals for the 0.05
```

```
            x += st
    return sorted(vals)

def _percent_to_int(s: str) -> int:
    """'Off' -> 0, '12%' -> 12"""
    return 0 if s == "Off" else int(s.rstrip("%"))

def _int_to_percent(v: int) -> str:
    """0 -> 'Off', 12 -> '12%'"""
    return "Off" if v == 0 else f"{int(v)}%"


# all allowed lists to be used by Widgets
ALLOWED_LRL = build_allowed_lrl(LRL_SEGMENTS)
ALLOWED_URL = build_allowed_url(URL_MIN, URL_MAX, URL_STEP)
PW_VALUES   = build_pw_values()
REF_VALUES  = list(range(REF_MIN, REF_MAX + 1, REF_STEP))


# ==========================
# 6) CUSTOM WIDGETS FOR PARAMETERS!!!!!!
# ==========================
class LRLSpinBox(QSpinBox):
    """Integer spinbox for LRL; enforces allowed piecewise values"""

    def __init__(self, parent=None):
        super().__init__(parent) # call base class constructor
        self.allowed = ALLOWED_LRL # creating self.allowed to use ALLOWED_LRL in this class
        self.setRange(min(self.allowed), max(self.allowed)) # min/max bounds keep in list
        self.setSuffix(" ppm") # add " ppm" after number
        self.setValue(60) # default value 60, this is also the nominal value

    def stepBy(self, steps: int) -> None:
        """
        called when user presses the up/down arrows; override it to jump within self.allowed
        so it won't just go +1/-1
        """
        current = self.value()
        try:
            idx = self.allowed.index(current) # find current position
        except ValueError:
            # if user typed in-between value, find the nearest index
            idx = min(range(len(self.allowed)), key=lambda i: abs(self.allowed[i] - current))
        idx = max(0, min(len(self.allowed) - 1, idx + steps))  # clamp to ends of list
```

```
        self.setValue(self.allowed[idx]) # set the new value

    def validate(self, text: str, pos: int):
        """
        controls whats accepted when you type in the box
        validator states!
        intermediate = allow typing to continue, so like 6 towards 60
        invalid = stops typing
        acceptable = what we want, in range
        """
        # strip the suffix to check number
        raw = text.replace(" ppm", "").strip()
        if raw == "":
            return (QValidator.Intermediate, text, pos) #space/empty, allow

        # if it's not an integer yet, let the user keep typing
        if not raw.lstrip("-").isdigit():
            return (QValidator.Intermediate, text, pos)

        val = int(raw)

        # while typing up to the minimum (ex 6 on the way to 60), allow it
        if val < self.minimum():
            return (QValidator.Intermediate, text, pos)

        # above maximum is invalid
        if val > self.maximum():
            return (QValidator.Invalid, text, pos)

        # inside bounds: only exact allowed values are acceptable
        if val in self.allowed:
            return (QValidator.Acceptable, text, pos)

        # within numeric bounds but not an allowed discrete value
        return (QValidator.Invalid, text, pos)

class URLSpinBox(QSpinBox):
    """SAME CONCEPT AS LRL!! except using allowed url"""

    def __init__(self, parent=None):
        super().__init__(parent)
        self.allowed = ALLOWED_URL # multiples of 5
        self.setRange(min(self.allowed), max(self.allowed))
        self.setSuffix(" ppm")
```

```
        self.setValue(120)

    def stepBy(self, steps: int) -> None:
        current = self.value()
        try:
            idx = self.allowed.index(current)
        except ValueError:
            # if user typed in-between value, find the nearest legal index
            idx = min(range(len(self.allowed)), key=lambda i: abs(self.allowed[i] - current)) # clamp to [0,
last]
        idx = max(0, min(len(self.allowed) - 1, idx + steps)) # new value
        self.setValue(self.allowed[idx])

    def validate(self, text: str, pos: int):
        # strip the suffix the widget appends
        raw = text.replace(" ppm", "").strip()
        if raw == "":
            return (QValidator.Intermediate, text, pos)

        # if it's not an integer yet, let the user keep typing
        if not raw.lstrip("-").isdigit():
            return (QValidator.Intermediate, text, pos)

        val = int(raw)

        # while typing up to the minimum (ex 6 on the way to 60), allow it
        if val < self.minimum():
            return (QValidator.Intermediate, text, pos)

        # above maximum is invalid
        if val > self.maximum():
            return (QValidator.Invalid, text, pos)

        # inside bounds: only exact allowed values are acceptable
        if val in self.allowed:
            return (QValidator.Acceptable, text, pos)

        # within numeric bounds but not an allowed discrete value
        return (QValidator.Invalid, text, pos)

class PWSpinBox(QDoubleSpinBox):
    """double spinbox for pulse width, 0.05 then 0.1–1.9 stepping"""
    # used by ModeEditorPage -> self.a/v_pw
    # values retrieved and saved in _collect_params()
```

```
    def __init__(self, parent=None):
        super().__init__(parent)
        self.allowed = PW_VALUES
        self.setRange(min(self.allowed), max(self.allowed))
        self.setDecimals(2) # show two decimals (0.05)
        self.setSingleStep(0.05)
        self.setSuffix(" ms")
        self.setValue(0.4) # nominal value

    def stepBy(self, steps: int) -> None: # same pattern
        """jump along our allowed values list rather than raw 0.05 steps"""
        cur = self.value()
        try:
            idx = self.allowed.index(cur)
        except ValueError:
            idx = min(range(len(self.allowed)), key=lambda i: abs(self.allowed[i] - cur))
        idx = max(0, min(len(self.allowed) - 1, idx + steps))
        self.setValue(self.allowed[idx])

    def validate(self, text: str, pos: int): # same pattern
        """accept only exact members of PW_VALUES"""
        try:
            val = float(text.replace(" ms", "").strip())
        except ValueError:
            return (QValidator.Intermediate, text, pos)
        if any(abs(val - a) < 1e-9 for a in self.allowed):
            return (QValidator.Acceptable, text, pos)
        if self.minimum() <= val <= self.maximum():
            return (QValidator.Invalid, text, pos)
        return (QValidator.Invalid, text, pos)

class RefPeriodSpinBox(QSpinBox):
    """ppinbox for ARP/VRP"""
    # no floats no validate . saved and loaded like other stuff
    # used by ModeEditorPage -> self.a/vrp
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setRange(REF_MIN, REF_MAX)
        self.setSingleStep(REF_STEP)
        self.setSuffix(" ms")
        self.setValue(250)
```

```
class AmplitudeWidget(QWidget):
    """
    widget w/ [Label] [Off/On dropdown] [Voltage spinbox]
    - If "Off" selected -> .value() returns "Off" (string)
    - If "On" selected  -> .value() returns a float snapped to valid step range

    A and V different ranges
    """
    # ModeEditorPage creates self.a_amp, self.v_amp = AmplitudeWidget("")
    # _collect_params() calls .value() on each
    # _apply_params_to_widgets() calls .setValue(...) on each

    def __init__(self, label_text: str,
                 min_v: float, max_v: float, step_v: float, default_v: float, # range and number shown when on
                 parent=None): # defining values
        super().__init__(parent)

        # remember range so snapping knows what to do
        self._min_v = float(min_v)
        self._max_v = float(max_v)
        self._step_v = float(step_v)

        # build a horizontal row: label | combo | spin
        row = QHBoxLayout(self) # 'self' is the container widget
        row.setContentsMargins(0, 0, 0, 0) # no extra padding inside

        self.label = QLabel(label_text) # ex "Atrial Amplitude:"
        self.state_combobox = QComboBox() # dropdown with "Off"/"On"
        self.state_combobox.addItems(["Off", "On"]) # add the two choices

        self.volt_spinbox = QDoubleSpinBox() # voltage input
        self.volt_spinbox.setDecimals(1) # show 1 decimal place
        self.volt_spinbox.setRange(self._min_v, self._max_v)  # chamber-specific range
        self.volt_spinbox.setSingleStep(self._step_v) # chamber-specific step
        self.volt_spinbox.setSuffix(" V") # add suffix
        self.volt_spinbox.setValue(default_v) # starting value

        # react when combo changes (disable/enable the spinbox)
        self.state_combobox.currentIndexChanged.connect(self._update_enabled_state) # ref to
_update_enabled_state
        self._update_enabled_state() # set initial enabled state

        # add all three widgets into the row
        row.addWidget(self.label)
```

```
        row.addWidget(self.state_combobox)
        row.addWidget(self.volt_spinbox)

    def _update_enabled_state(self) -> None:
        """enable the number field only if 'On' is selected"""
        self.volt_spinbox.setEnabled(self.state_combobox.currentText() == "On")

    def value(self):
        """return either 'Off' (str) or a float rounded to grid"""
        if self.state_combobox.currentText() == "Off":
            return "Off"

        v = float(self.volt_spinbox.value()) # whats in the box rn, can be anything, need to snap/modify
        # snap to the nearest limit defined by (min, step)
        steps = round((v - self._min_v) / self._step_v)
        snapped = round(self._min_v + steps * self._step_v, 1)
        # clamp
        snapped = max(self._min_v, min(self._max_v, snapped))
        return snapped

    ## ^^ basically whats happening is taking the current value - minimum and dividing by the step
    ## ex. current (2.87 - min 0.5)/0.1 step = 23.7 round to 24 steps
    ## then add to min and multiply by step to get actual value of 2.9

    def setValue(self, val) -> None:
        """set widget value from saved data: 'Off' or number volt"""
        if val == "Off":
            self.state_combobox.setCurrentText("Off")
        else:
            self.state_combobox.setCurrentText("On")
            self.volt_spinbox.setValue(float(val))

# ============================
# 7) screens/pages inside the stacked widget
# ============================
class LoginPage(QWidget):
    """
    registration + login page.
    - receive 'db' (Database) and 'on_login_ok' (callback) from MainWindow*
    - when login succeeds, we call on_login_ok(username) to tell MainWindow
    """

    def __init__(self, db: Database, on_login_ok, parent=None): # depends on db
        super().__init__(parent) # build the QWidget base
```

```
        self.db = db # remember the database so we can query it
        self.on_login_ok = on_login_ok  # remember callback to MainWindow *

        page = QVBoxLayout(self) # vertical page layout
        page.addWidget(QLabel("<h2>DCM — Welcome!</h2>"))  # simple title

        # registration group (top)
        reg_group = QGroupBox("New user registration")
        reg_form = QFormLayout(reg_group) # label : field pairs inside the group

        self.reg_name = QLineEdit() # name input
        self.reg_pass = QLineEdit() # password input
        self.reg_pass.setEchoMode(QLineEdit.Password) # hide characters

        RegisterButton = QPushButton("Register")# button the user clicks to register
        RegisterButton.clicked.connect(self._handle_register)  # connect signal -> slot (method below)

        reg_form.addRow("Name:", self.reg_name)
        reg_form.addRow("Password:", self.reg_pass)
        reg_form.addRow(RegisterButton)

        # login group (bottom)
        login_group = QGroupBox("Login")
        login_form = QFormLayout(login_group)

        self.login_name = QLineEdit()
        self.login_pass = QLineEdit()
        self.login_pass.setEchoMode(QLineEdit.Password)

        LoginButton = QPushButton("Login")
        LoginButton.clicked.connect(self._handle_login)

        login_form.addRow("Name:", self.login_name)
        login_form.addRow("Password:", self.login_pass)
        login_form.addRow(LoginButton)

        # add both groups to the page layout
        page.addWidget(reg_group)
        page.addWidget(login_group)
        page.addStretch(1) # spacing

    # slots (handlers) for the two buttons
    def _handle_register(self) -> None:
        """validate inputs, enforce limits, then add user to db"""
```

```
        name = self.reg_name.text().strip()  # .text() gets the text; .strip() trims spaces
        pw   = self.reg_pass.text()
        if not name or not pw:
            QMessageBox.warning(self, "Missing info", "Please enter a name and password!")
            return
        if self.db.user_count() >= MAX_USERS:
            QMessageBox.critical(self, "Max user capacity reached", f"Maximum of {MAX_USERS} users
stored.")
            return
        if not self.db.add_user(name, hash_password(pw)):  # returns False if duplicate or full
            QMessageBox.warning(self, "Registration failed!", "User exists or capacity reached.")
            return
        QMessageBox.information(self, "Success!", "Registration complete. Please log in.")
        self.reg_pass.clear()   # clear password field for safety

    def _handle_login(self) -> None:
        """check user and pass against db; if success, notify MainWindow."""
        name = self.login_name.text().strip()
        pw   = self.login_pass.text()
        if self.db.verify_user(name, hash_password(pw)):
            self.on_login_ok(name)# <- calls the callback passed from MainWindow
        else:
            QMessageBox.critical(self, "Login failed", "Invalid username or password.")


class DashboardPage(QWidget):
    """
    dashboard shows current device/comms state and mode buttons
    receive 'on_mode_click(mode)' callback to inform MainWindow which mode to open.
    """

    def __init__(self, on_mode_click, parent=None): # we call on_mode_click later in
MainWindow._open_mode_editor
        super().__init__(parent)
        self.on_mode_click = on_mode_click  # remember the callback

        page = QVBoxLayout(self) # vertical page layout
        page.addWidget(QLabel("<h2>Device Controller–Monitor</h2>"))

        # row of status labels
        status = QHBoxLayout()
        self.label_comms = QLabel("Comms: <b>Not Connected</b>")
        self.label_device = QLabel("Device: <i>None</i>")
        self.label_changed = QLabel("Status: <b>Last Device OK</b>")
```

```
        self.label_telemetry = QLabel("Telemetry: <b>OK</b>")
        for w in (self.label_comms, self.label_device, self.label_changed, self.label_telemetry):
            status.addWidget(w)
            status.addSpacing(20) # small space between labels
        status.addStretch(1)
        page.addLayout(status)

        # row of modes (AOO/VOO/AAI/VVI)
        row = QHBoxLayout()
        for mode in MODES:
            ModeButton = QPushButton(mode) # text/label is the mode
            ModeButton.setMinimumWidth(120) # make them wide enough
            # lambda captures 'mode' into 'm' so each button calls with its own value
            ModeButton.clicked.connect(lambda _, m=mode: self.on_mode_click(m))
            row.addWidget(ModeButton)
        row.addStretch(1)
        page.addLayout(row)
        page.addStretch(1)

    # public methods called by MainWindow to update the labels
    def show_comms(self, connected: bool) -> None:
        self.label_comms.setText(f"Comms: <b>{'Connected' if connected else 'Not Connected'}</b>")

    def show_device(self, device_id: str) -> None:
        self.label_device.setText(f"Device: <b>{device_id or 'None'}</b>")

    def show_changed(self, changed: bool) -> None:
        self.label_changed.setText("Status: <b>Device Changed</b>" if changed else "Status: <b>Last
Device OK</b>")

    def show_telemetry(self, state: str) -> None:
        mapping = {
            "ok": "Telemetry: <b>OK</b>",
            "out_of_range": "Telemetry: <b>Lost – Out of Range</b>",
            "noise": "Telemetry: <b>Lost – Noise</b>",
        }
        self.label_telemetry.setText(mapping.get(state, "Telemetry: <b>OK</b>"))

# EGRAM GRAPHS 3.2.5
class D1EgramView(QWidget):
    """
    real-time egram viewer with random data
    click start tart to begin and stop to pause
    """
```

```
def __init__(self, parent=None):
    super().__init__(parent)
    self.setMinimumHeight(200)
    self.timer = None # using qtimer, not running yet
    self.t = 0.0 # running time advanced on each tick
    self.dt = 0.02 # 0.02 s timestep or 50 Hz 1/0.02
    self.buffer_len = 400 # buffer 400 samples per trace
    self.atrialList = [] # empty lists for samples
    self.ventricularList = []
    self.surfaceList = []
    self.show_atrial = True # visibility
    self.show_ventricular = True
    self.show_surface = True

def start(self):
    if self.timer is None: # if no timer make one
        from PyQt5.QtCore import QTimer
        self.timer = QTimer(self)
        self.timer.timeout.connect(self._tick) # every time timer fires call _tick() to gen new data
        self.timer.start(int(self.dt * 1000)) # to ms

def stop(self):
    if self.timer: # if timer running stop
        self.timer.stop()
        self.timer.deleteLater()
        self.timer = None

def _tick(self):
    import math, random
    self.t += self.dt

    # generate random sinusoidal data (simple sinusoids + noise)
    atrial_data = 0.4 * math.sin(2 * math.pi * 1.5 * self.t) + 0.05 * random.uniform(-1, 1)
    ventricular_data = 0.8 * math.sin(2 * math.pi * 1.0 * self.t + 1.0) + 0.05 * random.uniform(-1, 1)
    surface_data = 0.6 * math.sin(2 * math.pi * 1.2 * self.t + 0.5) + 0.04 * random.uniform(-1, 1)

    self.atrialList.append(atrial_data) # append data to list
    self.ventricularList.append(ventricular_data)
    self.surfaceList.append(surface_data)

    if len(self.atrialList) > self.buffer_len:
        self.atrialList.pop(0)
        self.ventricularList.pop(0)
```

```
            self.surfaceList.pop(0)

        self.update() # trigger repaint

    def paintEvent(self, ev): # need to repaint
        from PyQt5.QtGui import QPainter, QPen, QColor
        from PyQt5.QtCore import QPointF, Qt

        p = QPainter(self)
        try:
            p.fillRect(self.rect(), self.palette().base()) # clear background using widget base colour

            w = self.width() # for layout math
            h = self.height()

            # 3 signal baselines
            row_h = h / 3.0 # divide widget into 3 horizontal bands, row_h each
            bases = [row_h * 0.5, row_h * 1.5, row_h * 2.5] # positions
            names = ["Atrial", "Ventricular", "Surface ECG"]
            colors = [QColor("red"), QColor("blue"), QColor("green")]
            series = [self.atrialList, self.ventricularList, self.surfaceList]
            shown = [self.show_atrial, self.show_ventricular, self.show_surface]

            for i in range(3): # for each bamd, set gray pen and draw horizontal baseline across width and lbel above baseline
                base = bases[i]
                p.setPen(QPen(Qt.gray))
                p.drawLine(0, int(base), w, int(base))
                p.drawText(5, int(base - 5), names[i])

                if not shown[i] or len(series[i]) < 2:
                    continue # skip drawing if trace hidden or not enough pts to draw line

                s = series[i] # list of samples
                step_x = w / max(1, len(s) - 1) # horizontal spacing between sequential points so the whole buffer spans the full width
                scale = row_h * 0.35
                pen = QPen(colors[i]) # design and colours
                pen.setWidth(2)
                p.setPen(pen)

                last = QPointF(0, base - scale * s[0]) # convert samples into pixels, x increase linear by step_x, y is base-scale*value. higher value above baseline
                for j in range(1, len(s)):
```

```
                pt = QPointF(j * step_x, base - scale * s[j])
                p.drawLine(last, pt)
                last = pt
        finally:
            p.end() # finalize


# ABOUT !!!
class AboutDialog(QDialog): #QDialog popup windows, vs QWidget for normal page
    """
    'About' panel listing model, software rev, serial, institution. 3.2.3
    """
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("About DCM")
        lay = QFormLayout(self)
        lay.addRow("Application model number:", QLabel(APP_MODEL_NUMBER))
        lay.addRow("Software revision:",       QLabel(APP_SOFTWARE_REV))
        lay.addRow("DCM serial number:",       QLabel(APP_SERIAL_NUMBER))
        lay.addRow("Institution name:",        QLabel(APP_INSTITUTION))
        buttons = QDialogButtonBox(QDialogButtonBox.Ok, parent=self)# ok button
        buttons.accepted.connect(self.accept)
        lay.addRow(buttons)


class SetClockDialog(QDialog): # popup
    """
    "The Set Clock function shall set the date and time of the device" 3.2.3
    """
    def __init__(self, current_DeviceTime: QDateTime, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Set Clock")
        lay = QFormLayout(self)

        self.dt_edit = QDateTimeEdit(current_DeviceTime) # picker widget internalized w current date/time
        self.dt_edit.setDisplayFormat("yyyy-MM-dd HH:mm:ss") # how it appears to user
        self.dt_edit.setCalendarPopup(True)

        lay.addRow("Device date/time:", self.dt_edit)

        buttons = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel, parent=self) # ok
and cancel, Q handles layout
        buttons.accepted.connect(self.accept)
        buttons.rejected.connect(self.reject)
```

```
        lay.addRow(buttons)

    def selected_datetime(self) -> QDateTime:
        return self.dt_edit.dateTime()


class ModeEditorPage(QWidget):
    """
    mode editor:
    - parameters tab: edit LRL, URL, A/V amplitude + width, ARP/VRP
    - summary tab: read-only summary of current values
    - egram (D2): placeholder text for next deliverable

    receives:
      db: Database     (for load/save)
      get_active_user: callable returning the username (or none)
    """

    def __init__(self, db: Database, get_active_user, parent=None):
        super().__init__(parent)
        self.db = db
        self.get_active_user = get_active_user
        self.current_mode: str = MODES[0] # default mode until changed

        # outer layout abd tavs
        page = QVBoxLayout(self)
        self.tabs = QTabWidget() # creates tabs along the top
        page.addWidget(self.tabs)

        # parameters tab
        self.tab_params = QWidget()
        self.tabs.addTab(self.tab_params, "Parameters")
        form = QFormLayout(self.tab_params) # two-column "Label : Widget"

        # choose mode at the top of the form
        self.mode_combo = QComboBox()
        self.mode_combo.addItems(MODES)
        self.mode_combo.currentTextChanged.connect(self.set_mode)  # when user changes, call
set_mode()
        form.addRow("Pacing Mode:", self.mode_combo)

        # widgets for each parameter
        self.lrl   = LRLSpinBox()
        self.url   = URLSpinBox()
```

```
# atrial: 0.5–3.2 V, step 0.1  (default 3.0 V)
self.atrial_amp = AmplitudeWidget("Atrial Amplitude:",
                    AMP_LOW_MIN, AMP_LOW_MAX, AMP_LOW_STEP, 3.0)

# ventricular: 3.5–7.0 V, step 0.5 (default 3.5 V)
self.ventricular_amp = AmplitudeWidget("Ventricular Amplitude:",
                    AMP_HIGH_MIN, AMP_HIGH_MAX, AMP_HIGH_STEP, 3.5)

self.atrial_pw  = PWSpinBox()
self.ventricular_pw  = PWSpinBox()
self.arp   = RefPeriodSpinBox()
self.vrp   = RefPeriodSpinBox()

# add them to the form with labels where appropriate
form.addRow("Lower Rate Limit (LRL):", self.lrl)
form.addRow("Upper Rate Limit (URL):", self.url)
form.addRow(self.atrial_amp)   # AmplitudeWidget includes its own label
form.addRow("Atrial Pulse Width:", self.atrial_pw)
form.addRow(self.ventricular_amp)
form.addRow("Ventricular Pulse Width:", self.ventricular_pw)
form.addRow("Atrial Refractory Period (ARP):", self.arp)
form.addRow("Ventricular Refractory Period (VRP):", self.vrp)

# hysteresis AAI/VVI only
self.hysteresis_state = QComboBox()
self.hysteresis_state.addItems(HYSTERESIS_STATES) # "Off" | "On"
self.HysRateLimit = LRLSpinBox() # use same choices as LRL
self.HysRateLimit.setEnabled(False) # only enabled when "On"

# when user flips Off/On, enable/disable HRL field
self.hysteresis_state.currentTextChanged.connect(
   lambda s: self.HysRateLimit.setEnabled(s == "On")
)

form.addRow("Hysteresis:", self.hysteresis_state)
form.addRow("Hysteresis Rate Limit (HRL):", self.HysRateLimit)

# rate Smoothing AAI/VVI only
self.smooth_up = QComboBox()
self.smooth_up.addItems(RATE_SMOOTH_CHOICES)
self.smooth_down = QComboBox()
self.smooth_down.addItems(RATE_SMOOTH_CHOICES)
form.addRow("Rate Smoothing Up:", self.smooth_up)
```

```
form.addRow("Rate Smoothing Down:", self.smooth_down)


# buttons: save / revert
buttons = QHBoxLayout()
self.SaveButton = QPushButton("Save Parameters")
self.RevertButton = QPushButton("Revert to Saved")
self.SaveButton.clicked.connect(self._handle_save) # connect click -> save
self.RevertButton.clicked.connect(self._handle_revert) # connect click -> revert
buttons.addWidget(self.SaveButton)
buttons.addWidget(self.RevertButton)
buttons.addStretch(1)
form.addRow(buttons)

# SUMMARY
self.tab_summary = QWidget()
self.tabs.addTab(self.tab_summary, "Summary")
summary_layout = QVBoxLayout(self.tab_summary)
self.label_summary = QLabel("")  # will show HTML text with values
summary_layout.addWidget(self.label_summary)
summary_layout.addStretch(1)

# EGRAM D2 TAB
self.tab_egram = QWidget()
self.tabs.addTab(self.tab_egram, "Egram (D2)")
egram_layout = QVBoxLayout(self.tab_egram)

# the viewer
self.egram_view = D1EgramView()
egram_layout.addWidget(self.egram_view)

# controls
EgramButtons = QHBoxLayout()
self.StartEgramButton = QPushButton("Start")
self.StopEgramButton  = QPushButton("Stop")
self.SaveEgramButton  = QPushButton("Save Strip (PNG)")
self.StartEgramButton.clicked.connect(self.egram_view.start)
self.StopEgramButton.clicked.connect(self.egram_view.stop)
self.SaveEgramButton.clicked.connect(self._save_egram_png)
EgramButtons.addWidget(self.StartEgramButton)
EgramButtons.addWidget(self.StopEgramButton)
EgramButtons.addWidget(self.SaveEgramButton)
EgramButtons.addStretch(1)
egram_layout.addLayout(EgramButtons)
```

```
        # initialize the page to the default mode (enable/disable fields + load saved if any)
        self.set_mode(self.current_mode)

        # after: self.tabs.addTab(self.tab_egram, "Egram (D2)")
        self.tabs.currentChanged.connect(self._on_tab_changed)

    def _on_tab_changed(self, idx: int) -> None:
        # auto start when the Egram tab is visible; stop when leaving
        if self.tabs.widget(idx) is self.tab_egram:
            self.egram_view.start()
        else:
            self.egram_view.stop()


    # public method: MainWindow calls this before showing the page
    def set_mode(self, mode: str) -> None: # remember newly selected mode
        """switch editor to a given mode; enable correct chamber fields and load saved data"""
        self.current_mode = mode
        # keep the combo box synchronized without re-triggering this method recursively, dont trigger
currentTextChanged
        if self.mode_combo.currentText() != mode:
            old = self.mode_combo.blockSignals(True)  # temporarily silence signals
            self.mode_combo.setCurrentText(mode) # set the visible value
            self.mode_combo.blockSignals(old) # restore signal behavior

        # enable only the chamber widgets that make sense for this mode, booleans true

        atrial_on = mode in ("AOO", "AAI")
        ventricular_on = mode in ("VOO", "VVI")
        inhibiting  = mode in ("AAI", "VVI")   # hysteresis + smoothing apply here

        #enabling amp depending on the modes
        self.atrial_amp.setEnabled(atrial_on); self.atrial_pw.setEnabled(atrial_on);
self.arp.setEnabled(atrial_on)
        self.ventricular_amp.setEnabled(ventricular_on); self.ventricular_pw.setEnabled(ventricular_on);
self.vrp.setEnabled(ventricular_on)

        # hysteresis + smoothing
        self.hysteresis_state.setEnabled(inhibiting) # on/enabled when inhibiting
        self.HysRateLimit.setEnabled(inhibiting and self.hysteresis_state.currentText() == "On")
        self.smooth_up.setEnabled(inhibiting)
        self.smooth_down.setEnabled(inhibiting)
```

```
        # load any saved params for the active user+mode; if none, keep current values
        self._handle_revert()

    # pulls current values from widgets into a plain dict ready for saving/export. at modes
    def _collect_params(self) -> Dict[str, Any]:
        return {
            "mode": self.current_mode,
            "LRL_ppm": self.lrl.value(),
            "URL_ppm": self.url.value(),
            "AtrialAmplitude_V": self.atrial_amp.value(),
            "AtrialPulseWidth_ms": round(self.atrial_pw.value(), 2),
            "VentricularAmplitude_V": self.ventricular_amp.value(),
            "VentricularPulseWidth_ms": round(self.ventricular_pw.value(), 2),
            "ARP_ms": self.arp.value(),
            "VRP_ms": self.vrp.value(),

            "Hysteresis": self.hysteresis_state.currentText(), # "Off"/"On"
            "HRL_ppm": self.HysRateLimit.value(), # used when Hysteresis == "On"
            "RateSmoothingUp_percent": _percent_to_int(self.smooth_up.currentText()),
            "RateSmoothingDown_percent": _percent_to_int(self.smooth_down.currentText()),
        }

    # apply a dict of params back onto the widgets, use for revert
    def _apply_params_to_widgets(self, p: Dict[str, Any]) -> None:
        self.lrl.setValue(p.get("LRL_ppm", self.lrl.value()))
        self.url.setValue(p.get("URL_ppm", self.url.value()))
        self.atrial_amp.setValue(p.get("AtrialAmplitude_V", self.atrial_amp.value()))
        self.atrial_pw.setValue(p.get("AtrialPulseWidth_ms", self.atrial_pw.value()))
        self.ventricular_amp.setValue(p.get("VentricularAmplitude_V", self.ventricular_amp.value()))
        self.ventricular_pw.setValue(p.get("VentricularPulseWidth_ms", self.ventricular_pw.value()))
        self.arp.setValue(p.get("ARP_ms", self.arp.value()))
        self.vrp.setValue(p.get("VRP_ms", self.vrp.value()))

        # NEW fields
        self.hysteresis_state.setCurrentText(p.get("Hysteresis", self.hysteresis_state.currentText()))
        # enable/disable HRL based on state
        self.HysRateLimit.setEnabled(self.hysteresis_state.currentText() == "On")
        self.HysRateLimit.setValue(p.get("HRL_ppm", self.HysRateLimit.value()))
        self.smooth_up.setCurrentText(_int_to_percent(p.get("RateSmoothingUp_percent",
_percent_to_int(self.smooth_up.currentText()))))
        self.smooth_down.setCurrentText(_int_to_percent(p.get("RateSmoothingDown_percent",
_percent_to_int(self.smooth_down.currentText()))))
```

```
    # EGRAM GRAPH PHOTOS 3.2.5
    def _save_egram_png(self) -> None:
        # self parent, dialog title, suggested default file name, file type filter only shows png returns selected
path, type
        path, _ = QFileDialog.getSaveFileName(self, "Save Egram Snapshot", "egram.png", "PNG Files
(*.png)")
        if not path:
            return
        screenshot = self.egram_view.grab()
        photoTaken = screenshot.save(path, "PNG")
        if photoTaken:
            QMessageBox.information(self, "Saved!", f"Egram snapshot saved to:\n{path}")
        else:
            QMessageBox.warning(self, "Error!", "Could not save the image.")


    # rebuild the HTML text for summary tab
    def _refresh_summary(self) -> None:
        p = self._collect_params()
        lines = [
            f"<b>Mode:</b> {p['mode']}",
            f"<b>LRL:</b> {p['LRL_ppm']} ppm",
            f"<b>URL:</b> {p['URL_ppm']} ppm",
            f"<b>Atrial:</b> Amp {p['AtrialAmplitude_V']} V, PW {p['AtrialPulseWidth_ms']} ms",
            f"<b>Ventricular:</b> Amp {p['VentricularAmplitude_V']} V, PW
{p['VentricularPulseWidth_ms']} ms",
            f"<b>ARP:</b> {p['ARP_ms']} ms, <b>VRP:</b> {p['VRP_ms']} ms",
        ]
        # hysteresis params for aai and vvi
        if self.current_mode in ("AAI", "VVI"):
            lines.append(f"<b>Hysteresis:</b> {p['Hysteresis']}"
                         + (f", HRL {p['HRL_ppm']} ppm" if p['Hysteresis']=='On' else ""))
            lines.append(f"<b>Rate Smoothing:</b> Up {_int_to_percent(p['RateSmoothingUp_percent'])}, "
                         f"Down {_int_to_percent(p['RateSmoothingDown_percent'])}")
        self.label_summary.setText("<br>".join(lines))

    def _defaults(self, mode: str) -> Dict[str, Any]:
        """general defaults based on nominal values, to refer back to"""
        base = {
            "LRL_ppm": 60,
            "URL_ppm": 120,
            "AtrialPulseWidth_ms": 0.40,
            "VentricularPulseWidth_ms": 0.40,
```

```
            "ARP_ms": 250,
            "VRP_ms": 320,
            "Hysteresis": "Off",
            "HRL_ppm": 60,
            "RateSmoothingUp_percent": 0,
            "RateSmoothingDown_percent": 0,
        }
        # only amplitudes depend on the mode, rest general
        base["AtrialAmplitude_V"]      = 3.0 if mode in ("AOO", "AAI") else "Off"
        base["VentricularAmplitude_V"] = 3.5 if mode in ("VOO", "VVI") else "Off"
        return base

    def _apply_defaults_for_mode(self, mode: str) -> None:
        """write defaults into the widgets (go back to when nothing saved)"""
        d = self._defaults(mode)
        self.lrl.setValue(d["LRL_ppm"])
        self.url.setValue(d["URL_ppm"])
        self.atrial_amp.setValue(d["AtrialAmplitude_V"])
        self.atrial_pw.setValue(d["AtrialPulseWidth_ms"])
        self.ventricular_amp.setValue(d["VentricularAmplitude_V"])
        self.ventricular_pw.setValue(d["VentricularPulseWidth_ms"])
        self.arp.setValue(d["ARP_ms"])
        self.vrp.setValue(d["VRP_ms"])
        self.hysteresis_state.setCurrentText(d["Hysteresis"])
        self.HysRateLimit.setEnabled(self.hysteresis_state.currentText() == "On")
        self.HysRateLimit.setValue(d["HRL_ppm"])
        self.smooth_up.setCurrentText(_int_to_percent(d["RateSmoothingUp_percent"]))
        self.smooth_down.setCurrentText(_int_to_percent(d["RateSmoothingDown_percent"]))


    # SAVE!!!!!!!!!!
    def _handle_save(self) -> None:
        user = self.get_active_user()
        if not user:
            QMessageBox.warning(self, "No user", "Please log in first.")
            return
        # safety: LRL should not exceed URL
        if self.lrl.value() > self.url.value():
            QMessageBox.warning(self, "Check Parameters!", "LRL must be <= URL.")
            return
        params = self._collect_params()
        self.db.save_params(user, self.current_mode, params)
        QMessageBox.information(self, "Saved", f"Parameters saved for {user} [{self.current_mode}].")
        self._refresh_summary()
```

```
    # REVERT!!!! (also called by set_mode)
    def _handle_revert(self) -> None:
        user = self.get_active_user()
        saved = self.db.load_params(user, self.current_mode) if user else {}
        if saved:
            self._apply_params_to_widgets(saved)
        else:
            # no saved params, go to default
            self._apply_defaults_for_mode(self.current_mode)
        self._refresh_summary()


# ==========================
# 8) Egram container (for D2)
# ==========================
class EgramData:
    """Egram page placeholder for streaming data in D2"""
    def __init__(self, time_ms: List[int], atrial_mv: List[float], ventricular_mv: List[float]):
        self.time_ms = time_ms # time in ms
        self.atrial_mv = atrial_mv # in mv
        self.ventricular_mv = ventricular_mv

    def __repr__(self) -> str: # printing in debug contexts
        return f"EgramData(n_samples={len(self.time_ms)})"


# ==========================
# 9) Main application window (owns pages + menus + status bar)
# ==========================
class MainWindow(QMainWindow):
    """
    MainWindow is the top-level frame:
      - creates a QStackedWidget to hold three pages
      - wires callbacks between pages (login -> dashboard -> editor)
      - hosts "Simulator" menu to flip comms/device/telemetry flags
      - shows status bar text based on those flags
    """

    def __init__(self):
        super().__init__()  # QMainWindow init
        self.setWindowTitle("DCM — Deliverable 1")
        self.resize(900, 600)
```

```
    # global app state
    self.db = Database(DB_FILE) # database instance pointing at json shared to children
    self.active_user: Optional[str] = None  # None until someone logs in

    # simulated states (D1: no real hardware)
    self.comms_connected = False
    self.device_id = ""
    self.device_changed = False
    self.telemetry_state = "ok"  # one of "ok" | "out_of_range" | "noise"

    # device clock 3.2.3 #2
    self.device_clock = QDateTime.currentDateTime()

    # creates a QStackedWidget (a deck where exactly one “page” is shown).
    self.stack = QStackedWidget()
    self.setCentralWidget(self.stack)

    # create the pages and pass callbacks/DB as needed
    self.page_login = LoginPage(self.db, self._on_login_ok)
    self.page_dash  = DashboardPage(self._open_mode_editor)
    self.page_edit  = ModeEditorPage(self.db, self._get_active_user) # active user?

    # add all pages to the stack
    for p in (self.page_login, self.page_dash, self.page_edit):
        self.stack.addWidget(p)

    self.stack.setCurrentWidget(self.page_login)  # start on login page

    # status bar at the bottom
    self.status_bar = QStatusBar()
    self.setStatusBar(self.status_bar)
    self._refresh_status_bar() # fills w initial status text

    # build top menu bar ("File", "Simulator")
    self._build_menus()

# build menus and wire actions to methods
def _build_menus(self) -> None:
    MenuBar = self.menuBar() # QMainWindow gives us a menu bar

    # file menu (export / reports / quit)
    menu_file = MenuBar.addMenu("File")
```

```
# reports submenu as per 3.2.4 1,2
menu_reports = menu_file.addMenu("Reports")
action_r1 = QAction("Bradycardia Parameters Report…", self)
action_r2 = QAction("Temporary Parameters Report…", self)
action_r1.triggered.connect(self._export_brady_params_report) # when clicked generate pdf
action_r2.triggered.connect(self._export_temp_params_report)
menu_reports.addAction(action_r1)
menu_reports.addAction(action_r2)

action_export = QAction("Export Saved Params JSON…", self) # export json
action_export.triggered.connect(self._export_all_json)
menu_file.addAction(action_export)

menu_file.addSeparator() # visual separator line in menu
action_quit = QAction("Quit", self)
action_quit.triggered.connect(self.close)
menu_file.addAction(action_quit)

# simulator menu (toggles and radio options for telemetry)
menu_sim = MenuBar.addMenu("Simulator")

# checkable action behaves like a checkbox in a menu
self.action_comms = QAction("Toggle Comms Connected", self, checkable=True)
self.action_comms.triggered.connect(self._toggle_comms)
menu_sim.addAction(self.action_comms)

self.action_changed = QAction("Toggle Device Changed", self, checkable=True)
self.action_changed.triggered.connect(self._toggle_device_changed)
menu_sim.addAction(self.action_changed)

self.action_set_device = QAction("Set Device ID…", self)
self.action_set_device.triggered.connect(self._set_device_id)
menu_sim.addAction(self.action_set_device)

# radio group for telemetry  ok default
self.telemetry_group = QActionGroup(self)
self.telemetry_group.setExclusive(True)  # makes them mutual-exclusive

self.action_tel_ok = QAction("Telemetry OK", self, checkable=True)
self.action_tel_oor = QAction("Loss: Out of Range", self, checkable=True)
self.action_tel_noise = QAction("Loss: Noise", self, checkable=True)
self.action_tel_ok.setChecked(True) # default selection

# utilities menu (about/set clock)
```

```
        menu_utilities = MenuBar.addMenu("Utilities")

        action_about = QAction("About…", self)
        action_about.triggered.connect(self._show_about)
        menu_utilities.addAction(action_about)

        action_clock = QAction("Set Clock…", self)
        action_clock.triggered.connect(self._set_clock_dialog)
        menu_utilities.addAction(action_clock)

        # put actions in the group + menu
        for a in (self.action_tel_ok, self.action_tel_oor, self.action_tel_noise):
            self.telemetry_group.addAction(a)
            menu_sim.addAction(a)

        # connect each radio action to a lambda that sets the string state
        self.action_tel_ok.triggered.connect(lambda: self._set_telemetry("ok"))
        self.action_tel_oor.triggered.connect(lambda: self._set_telemetry("out_of_range"))
        self.action_tel_noise.triggered.connect(lambda: self._set_telemetry("noise"))

    #file menu handler: export our JSON DB to a chosen file path
    def _export_all_json(self) -> None: # opens save file dialog, dest is full chosen path or empty if
canceled
        dest, _ = QFileDialog.getSaveFileName(self, "Export database JSON", "dcm_params.json")
        if not dest:  # user hit cancel
            return # exit
        with open(DB_FILE, "r", encoding="utf-8") as source, open(dest, "w", encoding="utf-8") as out:
            out.write(source.read())
        QMessageBox.information(self, "Exported", f"Saved to {dest}")
        # ^^ source: current database file on disk, out is file user picked in dialog

    # PNGS 3.2.4
    def _current_params(self) -> Optional[Dict[str, Any]]:
        """get currently shown editor parameters, or None if no user"""
        user = self._get_active_user()
        if not user:
            return None
        return self.page_edit._collect_params() # ask modeeditorpage to collect whats currently in widgets
and return dict

    def _report_header_html(self, report_name: str) -> str:
        # spec header fields
        now = QDateTime.currentDateTime().toString("yyyy-MM-dd HH:mm:ss") # get current time as
string
```

```
        return f"""
        <h2 style="margin-bottom:2px;">{report_name}</h2>
        <hr>
        <table cellspacing="4">
          <tr><td><b>Institution:</b></td><td>{APP_INSTITUTION}</td></tr>
          <tr><td><b>Printed:</b></td><td>{now}</td></tr>
          <tr><td><b>DCM Model/Version:</b></td><td>{APP_MODEL_NUMBER} /
{APP_SOFTWARE_REV}</td></tr>
          <tr><td><b>DCM Serial:</b></td><td>{APP_SERIAL_NUMBER}</td></tr>
          <tr><td><b>Device ID:</b></td><td>{self.device_id or 'None'}</td></tr>
          <tr><td><b>Report Name:</b></td><td>{report_name}</td></tr>
        </table>
        <br>
        """
    #^^ html string

    def _params_table_html(self, p: Dict[str, Any]) -> str:
        rows = []
        def row(k, v): rows.append(f"<tr><td><b>{k}</b></td><td>{v}</td></tr>")

        row("Mode", p["mode"])
        row("LRL", f'{p["LRL_ppm"]} ppm')
        row("URL", f'{p["URL_ppm"]} ppm')
        row("Atrial Amplitude", f'{p["AtrialAmplitude_V"]} V')
        row("Atrial Pulse Width", f'{p["AtrialPulseWidth_ms"]} ms')
        row("Ventricular Amplitude", f'{p["VentricularAmplitude_V"]} V')
        row("Ventricular Pulse Width", f'{p["VentricularPulseWidth_ms"]} ms')
        row("ARP", f'{p["ARP_ms"]} ms')
        row("VRP", f'{p["VRP_ms"]} ms')
        if p["mode"] in ("AAI", "VVI"):
            row("Hysteresis", p["Hysteresis"] + (f' (HRL {p["HRL_ppm"]} ppm)' if p["Hysteresis"]=="On"
else ""))
            row("Rate Smoothing Up", f'{p["RateSmoothingUp_percent"]}%')
            row("Rate Smoothing Down", f'{p["RateSmoothingDown_percent"]}%')

        return f"""
        <table border="1" cellspacing="0" cellpadding="4">
          {''.join(rows)}
        </table>
        """

    def _save_pdf(self, html: str, suggested: str) -> None:
        dest, _ = QFileDialog.getSaveFileName(self, "Save PDF", suggested, "PDF Files (*.pdf)")
        if not dest:
```

```
            return
        # print HTML to PDF
        doc = QTextDocument()
        doc.setHtml(html)
        printer = QPrinter(QPrinter.HighResolution)
        printer.setOutputFormat(QPrinter.PdfFormat)
        printer.setOutputFileName(dest)
        doc.print_(printer)
        QMessageBox.information(self, "Saved", f"PDF saved to:\n{dest}")

    def _export_brady_params_report(self) -> None:
        p = self._current_params()
        if not p:
            QMessageBox.warning(self, "No user", "Please log in and open the Mode Editor first.")
            return
        html = self._report_header_html("Bradycardia Parameters Report") + self._params_table_html(p)
        self._save_pdf(html, "Bradycardia Parameters Report.pdf")

    def _export_temp_params_report(self) -> None:
        p = self._current_params()
        if not p:
            QMessageBox.warning(self, "No user", "Please log in and open the Mode Editor first.")
            return
        html = self._report_header_html("Temporary Parameters Report") + self._params_table_html(p)
        self._save_pdf(html, "Temporary Parameters Report.pdf")


    # NAVIGATION CALLED BY LoginPage when login OK
    def _on_login_ok(self, username: str) -> None:
        self.active_user = username
        self.stack.setCurrentWidget(self.page_dash)  # switch to dashboard
        self.status_bar.showMessage(f"Logged in as {username}", 4000)

    # navigation: called by DashboardPage when a mode button clicked
    def _open_mode_editor(self, mode: str) -> None:
        self.page_edit.set_mode(mode) # tell editor which mode
        self.stack.setCurrentWidget(self.page_edit) # switch to editor page

    # helper passed to ModeEditorPage so it can ask who is logged in
    def _get_active_user(self) -> Optional[str]:
        return self.active_user

    # simulator handlers: flip internal flags and update labels/status
    def _toggle_comms(self) -> None:
```

```
        self.comms_connected = self.action_comms.isChecked()
        self.page_dash.show_comms(self.comms_connected)
        self._refresh_status_bar()

    def _toggle_device_changed(self) -> None:
        self.device_changed = self.action_changed.isChecked()
        self.page_dash.show_changed(self.device_changed)
        self._refresh_status_bar()

    def _set_device_id(self) -> None:
        # reuse a save dialog as a crude "enter a string" prompt.
        path, _ = QFileDialog.getSaveFileName(self, "Enter Device ID then Cancel or Save",
"device-1234.txt")
        if path:
            # get just filename without folder or extension
            self.device_id = os.path.splitext(os.path.basename(path))[0]
        self.page_dash.show_device(self.device_id)
        self._refresh_status_bar()

    def _set_telemetry(self, state: str) -> None:
        self.telemetry_state = state
        self.page_dash.show_telemetry(state)
        self._refresh_status_bar()

    # status bar
    def _refresh_status_bar(self) -> None:
        comms = "Connected" if self.comms_connected else "Not Connected"
        changed = "Device Changed" if self.device_changed else "Last Device OK"
        tel = {
            "ok": "Telemetry: OK",
            "out_of_range": "Telemetry: Lost – Out of Range",
            "noise": "Telemetry: Lost – Noise"
        }[self.telemetry_state]
        clock_str = self.device_clock.toString("yyyy-MM-dd HH:mm:ss")
        text = f"{comms} | Device: {self.device_id or 'None'} | {changed} | {tel} | Clock: {clock_str}"

        self.status_bar.showMessage(text)

    def _show_about(self) -> None:
        AboutDialog(self).exec_()

    def _set_clock_dialog(self) -> None:
        dlg = SetClockDialog(self.device_clock, self)
        if dlg.exec_() == QDialog.Accepted:
```

```
            self.device_clock = dlg.selected_datetime()
            self._refresh_status_bar()



# ==========================
# 10) RUNNING!!!!
# ==========================
if __name__ == "__main__":
    app = QApplication([]) # create the app object (one per process)

    win = MainWindow() # create main window
    win.show() # make it visible
    app.exec_() # enter Qt event loop (blocks until window closes)
```