Assignment 2 – Pacemaker Project

SFWRENG 3K04 – Software Development

James Cameron – camerj22 – 400450866

Ryan Brubacher – brubachr - 400457266

Anuja Perera - perera5 - 400459978

Matthew Rakic – rakicm1 - 400449728

Jack Vu – vuj23 - 400453031

Hunter Boyd – boydh4 - 400384654

ACADEMIC STATEMENT

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is our own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario.

November 29, 2024

Lab L01

Table of Contents

List of Figures

List of Tables

1. Introduction

This document outlines the comprehensive process of designing, developing, testing, and validating a pacemaker system. The project is divided into two main components to ensure a structured approach. The first component focuses on implementing Simulink Stateflows, which are used to model the eight operating modes of the pacemaker, representing the backend functionality of the pacemaker. These modes simulate the essential functions that the pacemaker must perform, allowing for control and accurate modelling of heart responses. The second involves developing a Device Controller Monitor (DCM), which includes a graphical user interface (GUI), representing the front-end portion of the pacemaker interface. This interface enables users to interact with the pacemaker, adjusting mode and given parameters as needed.

The primary objective of the project is to create a reliable and safe system that meets the critical requirements of a pacemaker operating in a life-critical environment. This detailed documentation is provided to ensure that every stage of the process is well-documented. This allows for traceability and simplifies the process of making future improvements, or modifications to the system. By following using structured approach, the system is designed to meet both current needs and easily adapt for the future.

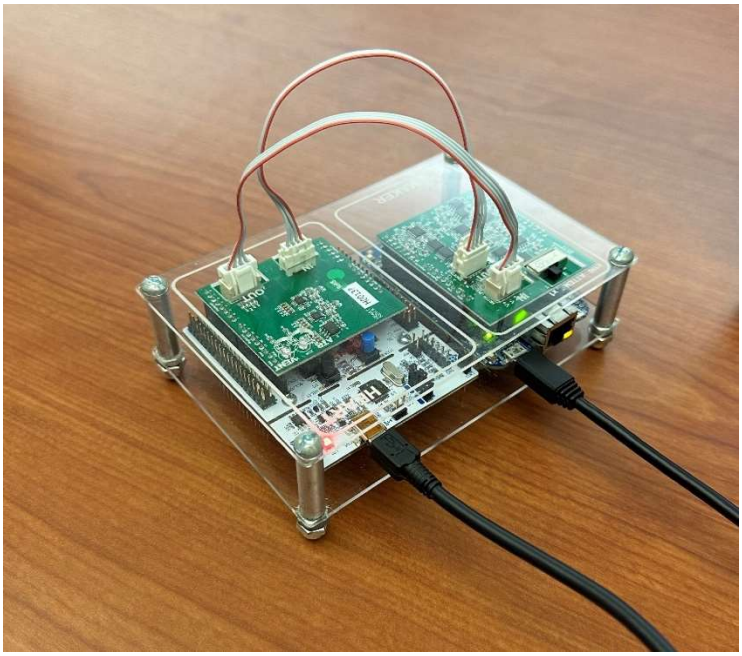


Figure 1: Pacemaker Board

2. Pacemaker

2.1 The Pacemaker

The pacemaker is a device designed to regulate abnormal heartbeats by delivering electrical pulses to the heart at a normal rate. The AOO and VOO modes are designed to deliver fixed rate pacing to the heart without monitoring for natural heartbeats, ensuring constant pacing. In AAI and VVI modes, the pacemaker will sense atrial or ventricular activity in the heart and will only deliver electrical pulses when no natural beat is detected. The rate adaptive modes follow the same design as the previous modes but track physical activity from the user through the onboard accelerometer and adjust pacing rates. These are the eight modes required to be implemented in assignment 2.

2.2 Requirements

The following requirements govern the development of the pacemaker operating modes in Simulink:

- **Modes**: The system must support AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR modes. These should be hidden in a subsystem.
- **Programmable Parameters Subsystem**: The system should allow setting pulse characteristics (pulse width and amplitude) and rate characteristics (lower and upper rate limits, delays).
- **Chamber Pacing**: The design must specify whether the atrium, ventricle, or both chambers are being paced, with appropriate sensing or pacing for inhibited modes (AAI, VOOR).
- **Hardware Abstraction**: Use a Simulink subsystem to abstract the hardware (pins and interfaces) from the logic, allowing easy modification of pin mapping without affecting the core model.
- **Serial Communication**: Through UART communication, have the DCM and Simulink system communicate by sending and receiving data to each other.

In this assignment, eight therapy modes are used, AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR where a limited number of parameters are used. Shown in Table 1 are the parameters used for these modes.

Table 1: Programmable Parameters for Bradycardia Therapy Modes

Parameter	Nominal	AOO	VOO	AAI	VVI	AOOR	VOOR	AAIR	VVIR
Lower Rate Limit	60 ppm (30 – 50) (50 – 90) (90 – 175)	X	X	X	X	X	X	X	X
Upper Rate Limit	120 ppm (50 – 175)	X	X	X	X	X	X	X	X
Atrial Amplitude	3.5 V (0.5 – 3.2)	X		X		X		X	
Ventricular Amplitude	3.5 V (3.5 – 7.0)		X		X		X		X
Atrial Pulse Width	0.4 ms (0.05) (0.1 – 1.9)	X		X		X		X	
Ventricular Pulse Width	0.4 ms (0.05) (0.1 – 1.9)		X		X		X		X
Atrial Sensitivity	0.75 mV (0.25, 0.5, 0.75, 1.0-10)			X				X	
Ventricular Sensitivity	2.5 mV (0.25, 0.5, 0.75, 1.0)				X				X
VRP	320 ms (150-500)				X				X

ARP	250 ms (150-500)			X				X	
PVARP	250 ms (150-500)			X				X	
Hysteresis	Off (Off or LRL)			X	X				X
Rate Smoothing	Off (Off, 3, 6, 9, 12, 18, 21, 25%)			X	X			X	X
Maximum Sensor Rate	120 ppm (50 - 175)					X	X	X	X
Activity Threshold	Med (8) (0 - 16)					X	X	X	X
Reaction Time	30 sec (10 - 50)					X	X	X	X
Response Factor	8 (1 - 16)					X	X	X	X
Recovery Time	5 min (2 - 16)					X	X	X	X

2.3 Design Decisions

Stateflow Charts: Each pacemaker mode (AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, VVIR) is designed as a dedicated subchart within the main Simulink Stateflow chart to provide modular and independent control over the operation modes. By creating distinct subcharts for each mode, the design enables clear separation of functionality, making it easier to test, modify,

and expand specific modes as required. Each state transition is controlled by well-defined input signals, such as sensing events or rate limits, ensuring that each mode operates based on its specific parameters without interference from other modes.

Pulse Characteristics: Adjustable pulse parameters, including pulse width and amplitude, are integrated as input variables to the Stateflow chart, allowing the pacemaker's output characteristics to be configured dynamically. This approach makes it possible to customize pacing settings for different scenarios or patient requirements directly through the input parameters. These parameters are transmitted via UART to the Digital Controller Module (DCM), allowing seamless adjustments through the user interface.

Hardware Hiding: A separate subsystem was introduced for pin mapping to facilitate hardware abstraction. By isolating the hardware-specific components, such as input and output pin mappings, in a dedicated subsystem, the design enables adaptability across different hardware configurations without the need to alter the primary state chart. This approach to hardware hiding simplifies future updates to the hardware setup, allowing the underlying logic of the pacemaker modes to remain unchanged even if new hardware components are introduced.

Safety Considerations: Clear and detailed annotations are included throughout the model to ensure traceability and understanding of the design. Each key component, state, and transition is meticulously labeled, and descriptions are provided to clarify functionality and operation. To enhance safety, the system is designed to enter an initial "off" mode when powered on, ensuring no unintended actions occur and allowing the system to stabilize before any operations are executed. This initial safety state provides an added safeguard against accidental activations and allows the system to verify readiness. Furthermore, for UART communication, a synchronization code is employed to validate proper communication before processing subsequent commands, ensuring reliable operation. This organized and safety-conscious approach also improves maintainability, making it easier for future engineers or users to review, interpret, and validate the model while ensuring compliance with safety standards.

2.4 Simulink Model Overview

2.4.1 Main Simulink Model

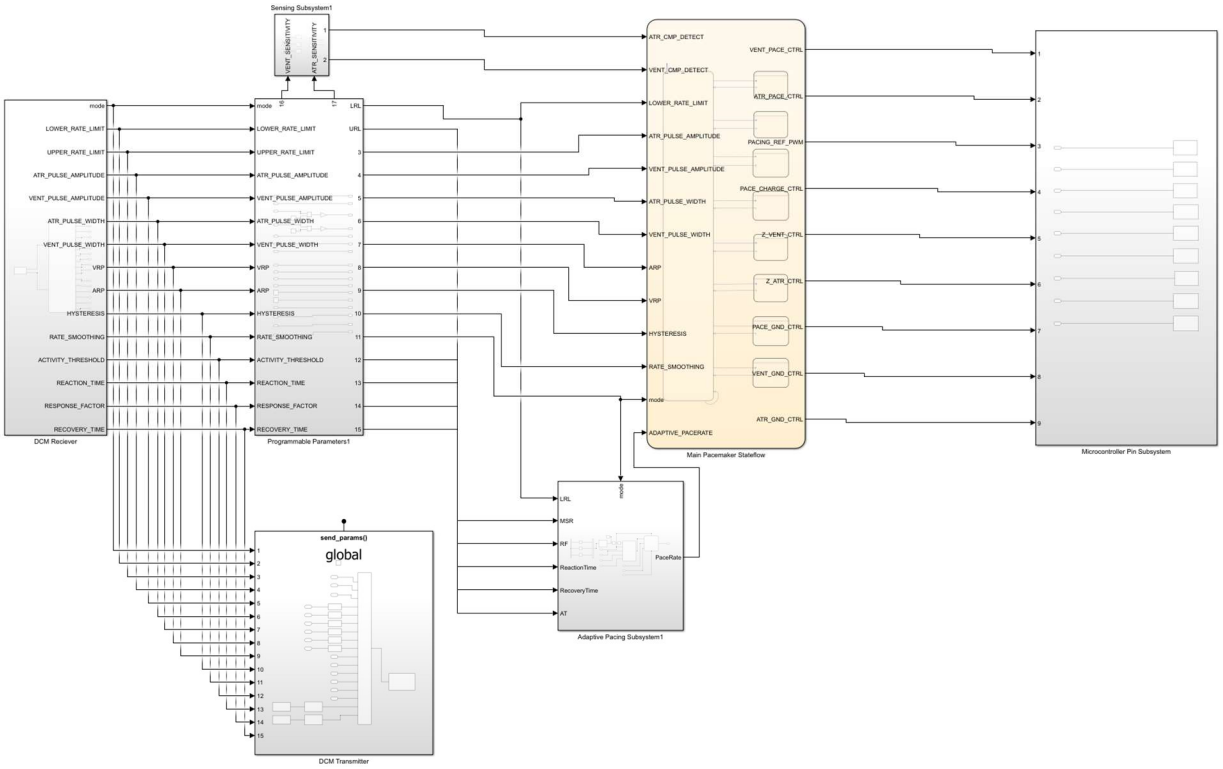


Figure 2: Main Simulink Model for Pacemaker

The main Simulink model is composed of six subsystems and one main Stateflow chart. Each subsystem handles different functional aspects of the pacemaker, including sensing, programmable parameters, rate calculations for rate adaptive pacing, UART receiving and UART transmitting, and hardware pin mapping. The Stateflow chart contains the core logic of the pacemaker, governing the operational modes (AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, VVIR) and transitions between them based on inputs from the subsystems. This modular design allows for flexibility and ease of updating specific components without impacting the entire model.

2.4.2 Programmable Parameters Subsystem

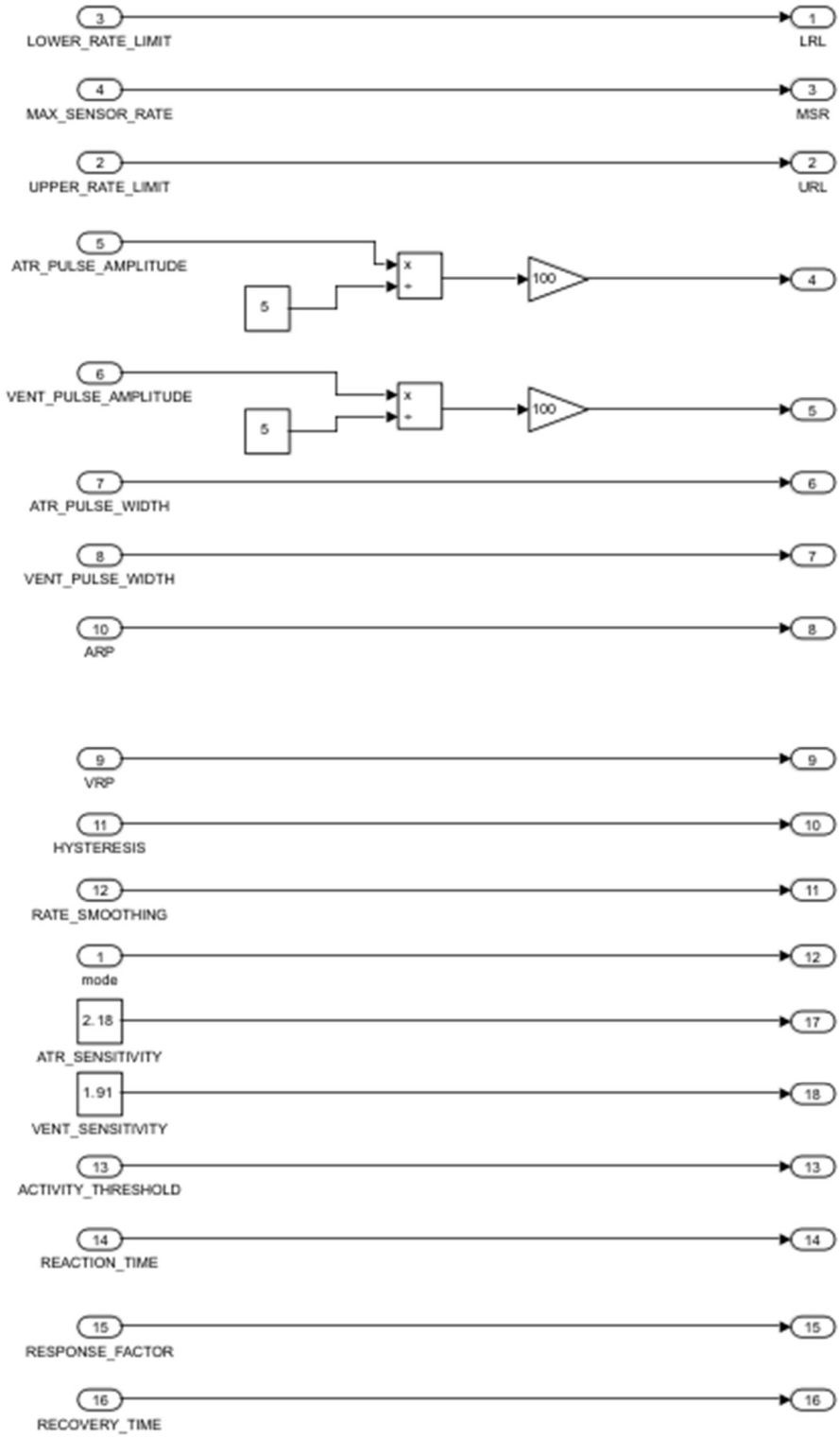


Figure 3: Programmable Parameter Subsystem

The subsystem shown above manages the programmable parameters for the pacemaker. It allows the user to set values related to the four primary modes of operation—AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR. Key parameters include the lower rate limit, upper rate limit, atrium and ventricle pulse widths, pulse amplitudes, sensitivity levels, and refractory periods (ARP and VRP). The subsystem includes inputs for mode selection, as well as parameters designed for future operational modes, such as Boolean inputs for hysteresis and rate smoothing, along with a value for the PVARP. Additionally, inputs such as activity threshold, reaction time, recovery time, maximum sensor rate and response factor are used for the rate calculations in rate adaptive modes.

2.4.3 Sensing Subsystem

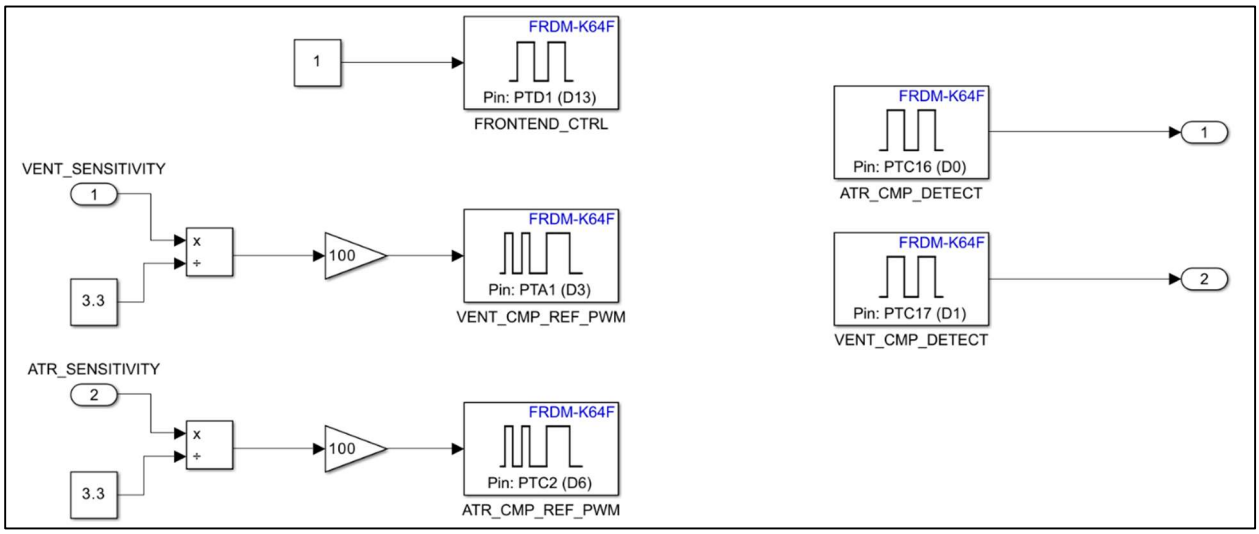


Figure 4: Sensing Subsystem

The diagram above shows the sensing subsystem for the pacemaker. This subsystem processes the inputs VENT_SENSITIVITY and ATR_SENSITIVITY by dividing each by the maximum capacitor voltage (3.3V), then applying a gain of 100 to set the duty cycle used to charge the comparison voltage. These processed values are then directed to the appropriate pins for further

use in the sensing circuit. The outputs from this subsystem are VENT_CMP_DETECT and ATR_CMP_DETECT, that output when activity is sensed from the corresponding lead.

The decision to isolate the sensing subsystem from the main Stateflow chart was motivated by a hardware abstraction approach. Since both the PWM generation values and input pins are hardware-dependent, isolating them allows for easier modifications if the hardware changes. By encapsulating this functionality in a separate subsystem, any changes in hardware would require adjustments only within this subsystem, without affecting the main logic in the Stateflow. This design enhances modularity and simplifies future hardware adaptations.

2.4.4 Microcontroller Pin Mapping Subsystem

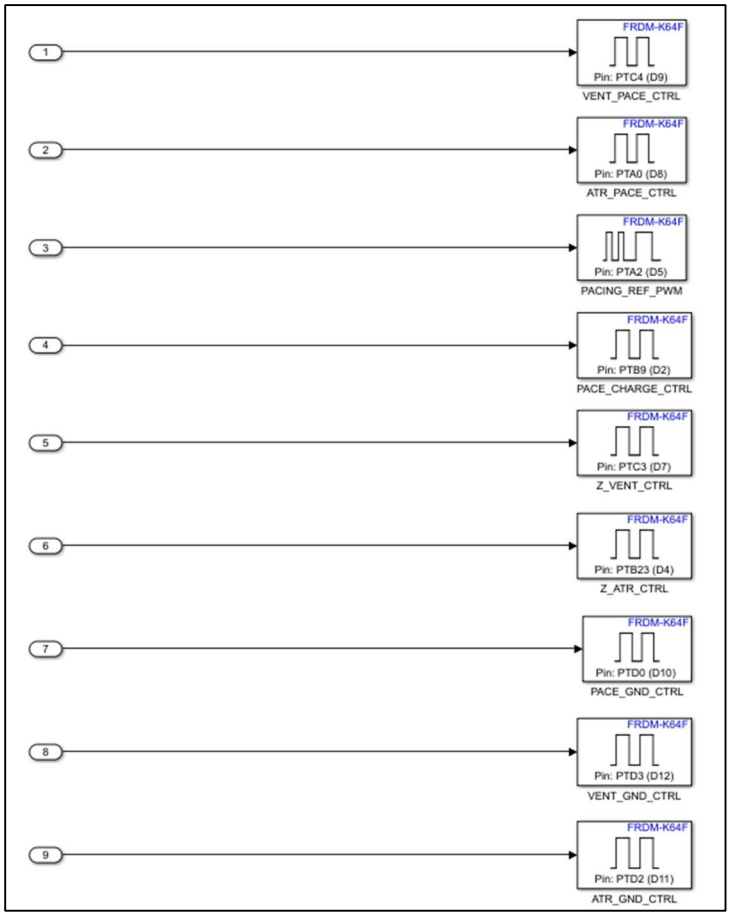


Figure 5: Microcontroller Pin Mapping Subsystem

The subsystem above maps the outputs from the main pacemaker Stateflow to the appropriate pins in the microcontroller. It is implemented in a separate subsystem to employ hardware hiding, making it so that this design isolates hardware-specific elements from the core logic of the pacemaker. By having the pin mapping and hardware-dependent components in their own subsystem, the main Stateflow logic remains independent of the underlying hardware. This approach enhances modularity and makes the system more adaptable to future hardware changes, as only this subsystem would need to be updated rather than modifying the entire Stateflow.

2.4.5 DCM Receiver

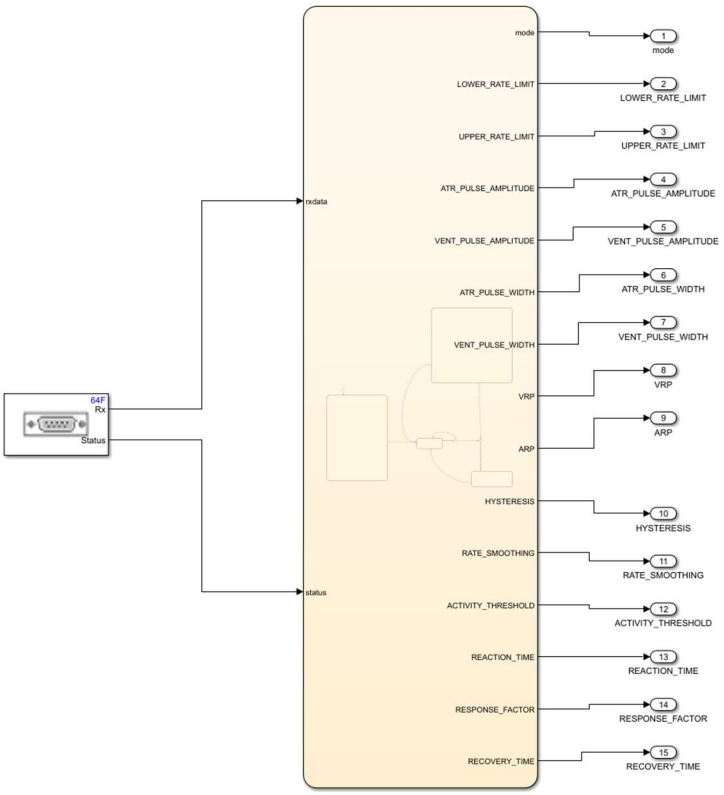


Figure 6: DCM Receiver subsystem

The DCM Receiver subsystem, outlined in Figure 6, describes how our Simulink system receives and translates data from the DCM for use within Simulink. By receiving and unpacking the data, it becomes readable and ready for processing. The outputted data can now be used in systems such as our main state flow chart, DCM transmitter, and programmable parameters.

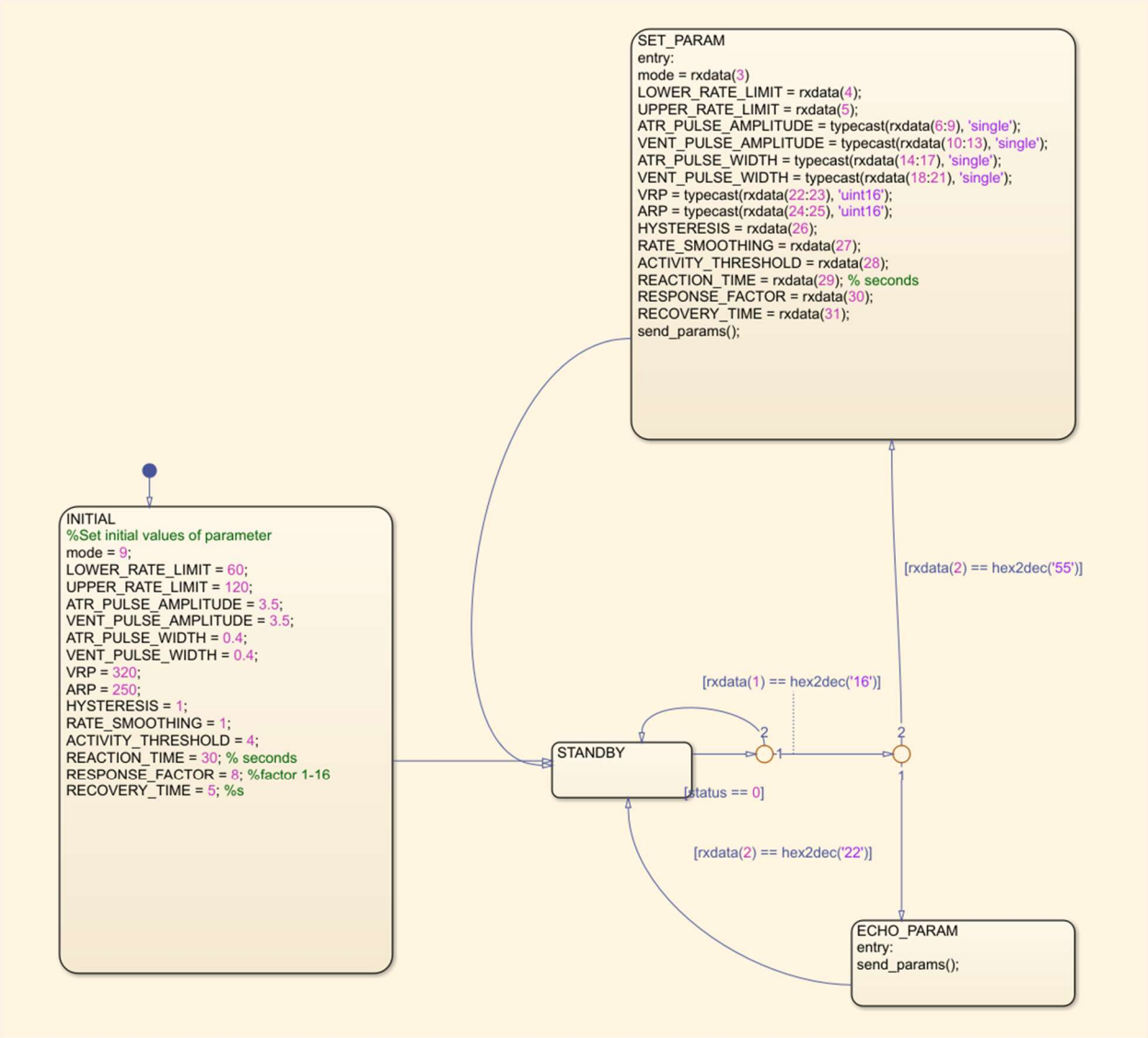


Figure 7: DCM Receiver state flow chart

The DCM Receiver Stateflow demonstrates our data receiving and unpacking processes. Starting in the initial state, we set our mode to 9 (off) and default on the parameters to prevent any bradycardia therapy from occurring. The system continually waits until data is received from the DCM and is unpacked into its respective byte sizes (Ie. ‘double’, ‘single, ‘uint8’). This data is set as outputs which can be used throughout the Simulink system as input parameters. Lastly, a send_params function is used as a global trigger, telling the DCM transmitter to relay this data back to the DCM, confirming if the correct data was originally received.

2.4.6 DCM Transmitter Subsystem

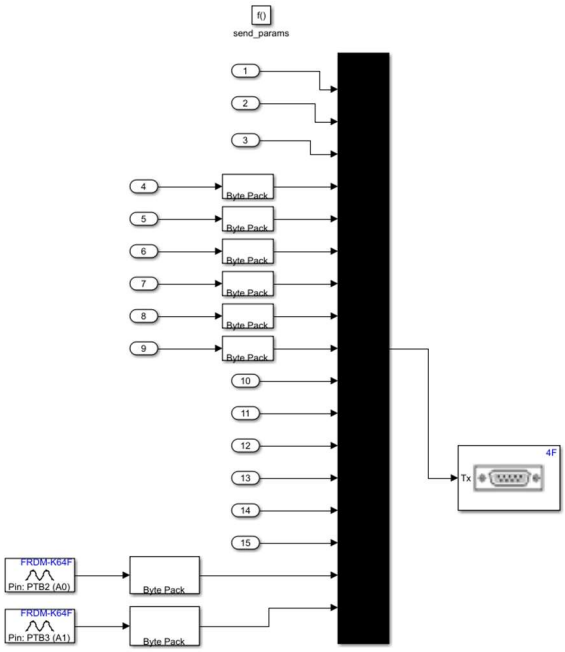


Figure 8: DCM Transmitter Subsystem

The DCM transmitter subsystem multiplexes the DCM Receiver outputs and sends them back to the DCM. Ports 4 through 9 are packed because they contain single and double values, which require compression to reduce packet size. The final two ports, which contain the Egram data for the atrium and ventricle signals, are also packed since they are used in the DCM. The remaining ports are transmitted without packing as they consist of uint8 byte values. The subsystem also contains the send_params() function call here, which triggers and sends data when it receives a signal from the DCM receiver Stateflow.

2.4.7 Adaptive Pacing Subsystem

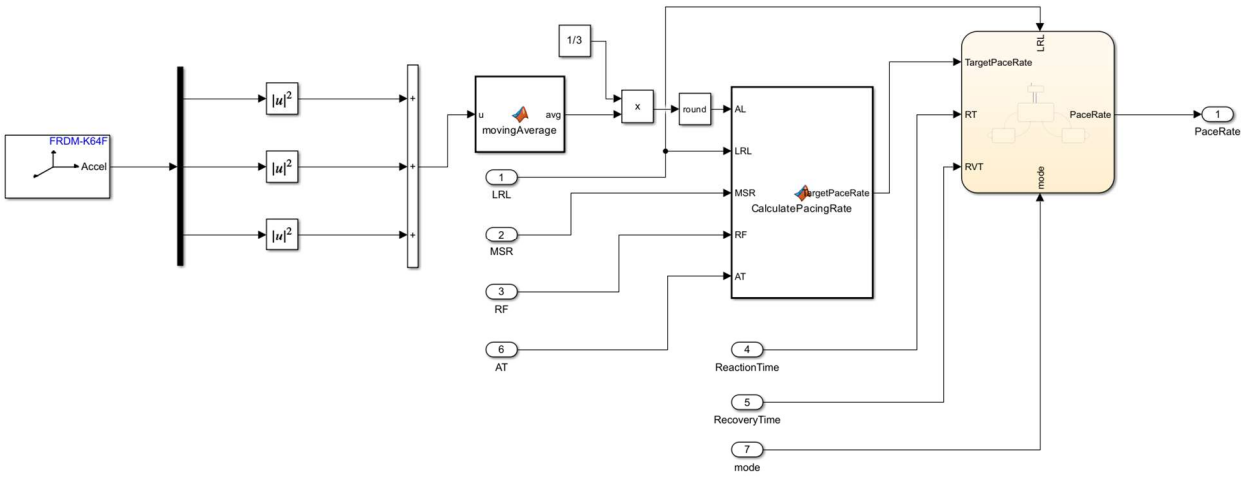


Figure 9: Adaptive Pacing subsystem

The adaptive pace rate subsystem first processes accelerometer data by receiving the raw x, y, and z components. These components are passed through a bus where they are individually squared, effectively mapping their original range of -4 to 4 into a positive range of 0 to 16. The squared values are then summed to obtain a combined activity measure, which is subsequently passed through a moving average filter. This filter smooths the data, helping to generate a stable, real-time output.

```
function avg = movingAverage(u)
    % Persistent variables to store buffer and index
    persistent buffer index count;

    % Number of samples for 1-second average
    N = 20;

    % Initialize buffer and index on first call
    if isempty(buffer)
        buffer = zeros(1, N); % Circular buffer
        index = 1;            % Current write position
        count = 0;            % Number of values added
    end

    % Add new sample to the buffer
    buffer(index) = u;
    index = mod(index, N) + 1; % Update index (wraps around)

    % Update count of samples (max is N)
    count = min(count + 1, N);

    % Calculate average (sum only the valid elements)
    avg = sum(buffer) / count;
end
```

Figure 10: Rate Smoothing Function

The moving average filter is designed with a buffer of size 20, reflecting the 200Hz sampling frequency of the accelerometer. As new data is received, it fills the buffer, and once the buffer reaches its maximum capacity, a wraparound mechanism is employed to overwrite the oldest data. The filter then calculates the sum of all values within the buffer and returns the average.

```
function TargetPaceRate = CalculatePacingRate(AL, LRL, MSR, RF, AT)

    if AL < AT
        TargetPaceRate = LRL;
    else
        TargetPaceRate = min(LRL + (AL - AT) * RF, MSR);
    end
end
```

Figure 11: Pacing Rate Function

To calculate the target pacing rate based on the patient's activity level, the output from the moving average filter is first divided by 3 and rounded to scale the value back into the range of 0 to 16. This scaled value is then used in the *CalculatePacingRate* function to determine the appropriate pacing rate.

The function operates as follows:

- **Activity Below Threshold**: If the activity level is below the predefined activity threshold, the target pacing rate is set to the lower rate limit, ensuring that the pacemaker maintains a minimum pace regardless of low activity.
- **Activity Above Threshold**: When the activity level exceeds the threshold, the target pacing rate is calculated by taking the minimum of two values:
    - The lower rate limit plus the difference between the current activity level and the activity threshold, adjusted by a response factor.
    - The maximum sensor rate, which serves as an upper limit for pacing.

This approach allows the pacing rate to dynamically adjust based on the patient's activity while ensuring that it stays within safe operational limits.

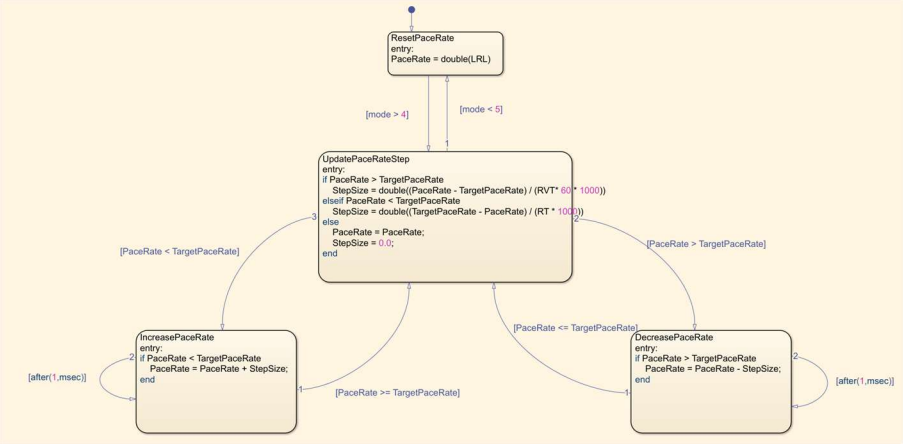


Figure 12: Updating Pacing Rate Stateflow

The Stateflow logic calculates the required step size to adjust the pacing rate based on two key parameters: the reaction time and recovery time. It then transitions to the appropriate state, where the current pacing rate is either increased or decreased by the calculated step size until it reaches the target pacing rate.

- **Rate Adjustment:** If the target pacing rate is higher than the current pace rate, the system will incrementally increase the pacing rate, and if the target is lower, the rate will be gradually decreased. The step size ensures that these adjustments occur smoothly and at a rate that is responsive but not overly abrupt.
- **Default State:** In addition to the dynamic rate adjustment, there is a default state that sets the pace rate to the lower rate limit. This state is particularly useful when first transitioning into a rate adaptive mode, ensuring that the pacemaker operates within a safe minimum rate before adaptive adjustments begin.

This approach ensures a controlled and responsive pacing adjustment, allowing the pacemaker to adapt to varying activity levels while avoiding sudden changes that could be detrimental to the patient.

2.4.8 Main Stateflow Chart

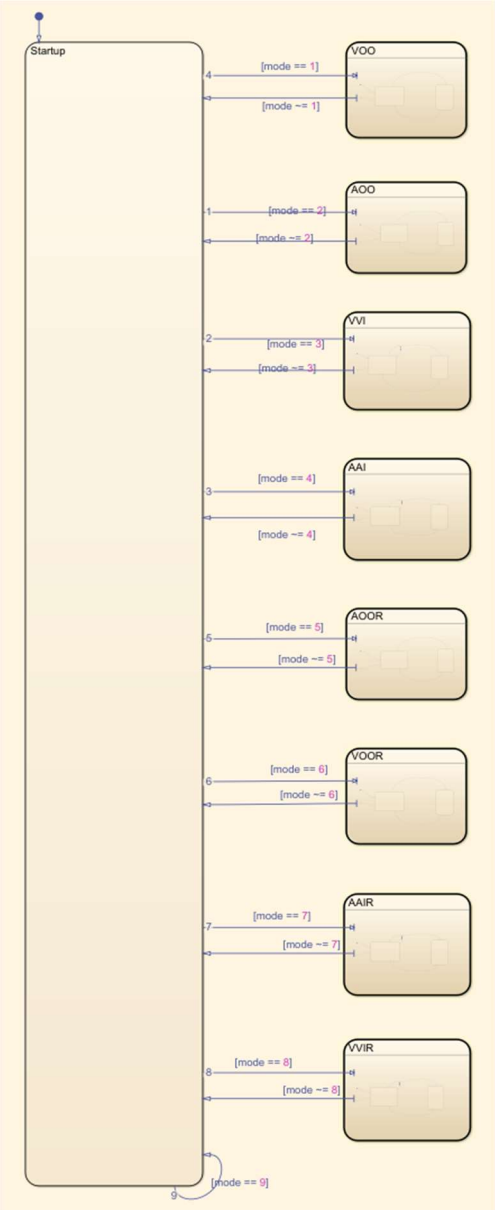


Figure 13: Main Stateflow Chart

The primary Stateflow chart establishes the bradycardia mode for pacemaker operation, with the mode set as an input in the Input Parameter System (Figure 3). Initially, the chart enters a “Startup” state before transitioning to the bradycardia state specified by one of 9 modes (modes 1-9). Currently, only a single mode can be active per build, with the pacemaker remaining in that mode until reconfiguration. Future enhancements will support dynamic mode switching,

allowing users to select a mode via a switch input, which will prompt a seamless state transition within the chart. The “Startup” mode can also be altered as future features are implemented.

2.4.9 VOO Stateflow

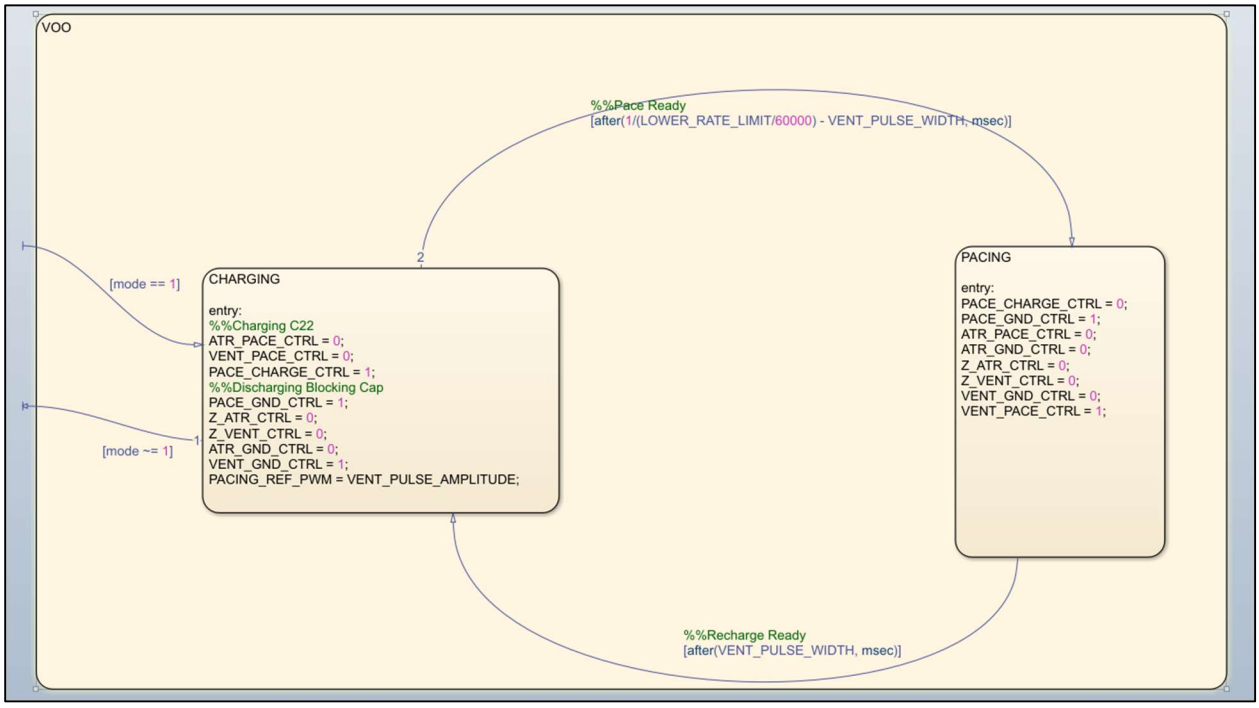


Figure 14: Nested VOO mode Stateflow

VOO Mode Overview: The VOO mode provides constant, asynchronous pacing to the ventricle at a rate determined by the lower rate limit, regardless of the natural heartbeat. The pacemaker delivers pulses at this rate, ensuring consistent stimulation without any heartbeat sensing.

Charging/Discharging State: The charging/discharging state controls the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined VENT_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, allowing the capacitor to discharge.

Pacing State: The pacing state controls the delivery of the pacing pulse to the ventricle. When VENT_PACE_CTRL is set high, capacitor C22 discharges into the ventricle, delivering a pacing

pulse. This also charges the blocking capacitor C21, ensuring that current flows only in the intended direction.

Stateflow Timing: The state transitions from charging to pulsing based on the lower rate limit, which is input in pulses per minute and divided by 60000 to yield pulses per millisecond. Dividing 1 by this value gives the total period. The state switches from charging to pulsing after a duration equal to the period minus the pulse width, then back to charging after the pulse width time. Upon each state entry, the output controls adjust according to the operation requirements.

2.4.10 AOO Stateflow

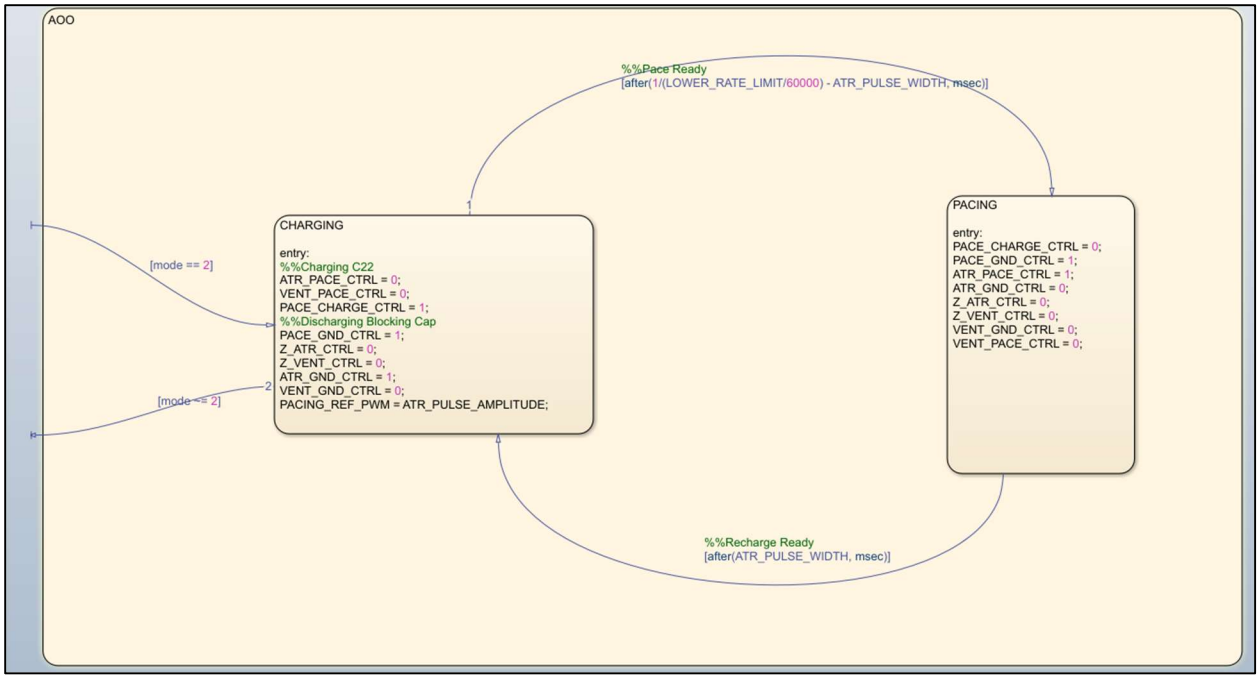


Figure 15: Nested AOO Stateflow

AOO Mode Overview: The AOO mode provides constant, asynchronous pacing to the atrium at a rate determined by the lower rate limit, regardless of the natural heartbeat. The pacemaker delivers pulses at this rate, ensuring consistent stimulation without any heartbeat sensing.

Charging/Discharging State: The charging/discharging state controls the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined ATR_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To

discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, allowing the capacitor to discharge.

Pacing State: The pacing state controls the delivery of the pacing pulse to the atrium. When ATR_PACE_CTRL is set high, capacitor C22 discharges into the atrium, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring that current flows only in the intended direction.

Stateflow Timing: The state transitions from charging to pulsing based on the lower rate limit, which is input in pulses per minute and divided by 60000 to yield pulses per millisecond. Dividing 1 by this value gives the total period. The state switches from charging to pulsing after a duration equal to the period minus the pulse width, then back to charging after the pulse width time. Upon each state entry, the output controls adjust according to the operation requirements.

2.4.11 VVI Stateflow

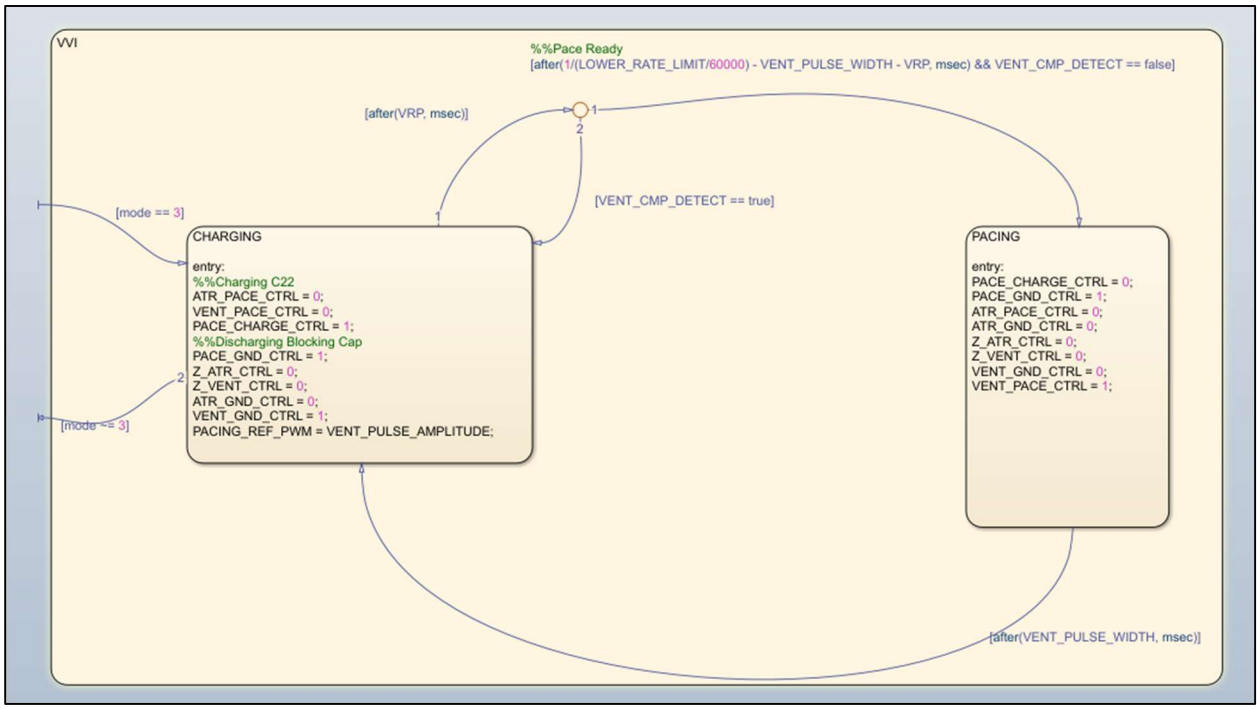


Figure 16: Nested VVI Stateflow

VVI Mode Overview: The VVI mode provides ventricular pacing at a rate determined by the lower rate limit, with the added capability to sense natural ventricular activity and inhibit pacing

when appropriate. This mode outputs pulses at the specified rate only when no intrinsic heartbeat is detected, ensuring pacing support when needed and preventing unnecessary stimulation.

Sensing: In VVI mode, ventricular sensing is employed to determine the appropriate timing for pacing. This process relies on the output from the VENT_CMP_DETECT pin within the sensing subsystem. The threshold voltage for comparison is established based on the input from VENT_SENSITIVITY, which is scaled to generate a PWM signal at VENT_CMP_REF_PWM to charge the comparison capacitor.

Charging/Discharging State: The charging/discharging state controls the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined VENT_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, allowing the capacitor to discharge.

Pacing State: The pacing state controls the delivery of the pacing pulse to the ventricle. When VENT_PACE_CTRL is set high, capacitor C22 discharges into the ventricle, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring that current flows only in the intended direction.

Stateflow Timing: The transition between states from charging to pulsing is governed by the lower rate limit, specified in pulses per minute and converted to pulses per millisecond by dividing by 60,000. The inverse of this value determines the total period. The system switches from the charging state to a junction state after a duration defined by the ventricular refractory period (VRP). At this junction, if the VENT_CMP_DETECT input pin reads high, the state reverts to charging, allowing for the VRP duration to elapse. Conversely, if no intrinsic activity is detected, the state transitions to pulsing after a duration equal to the total period minus the pulse width and VRP, returning to charging following the pulse width interval. This is to ensure the paces are delivered at the specified rate. With each state entry, the output controls are adjusted to align with the operational requirements.

2.4.12 AAI Stateflow

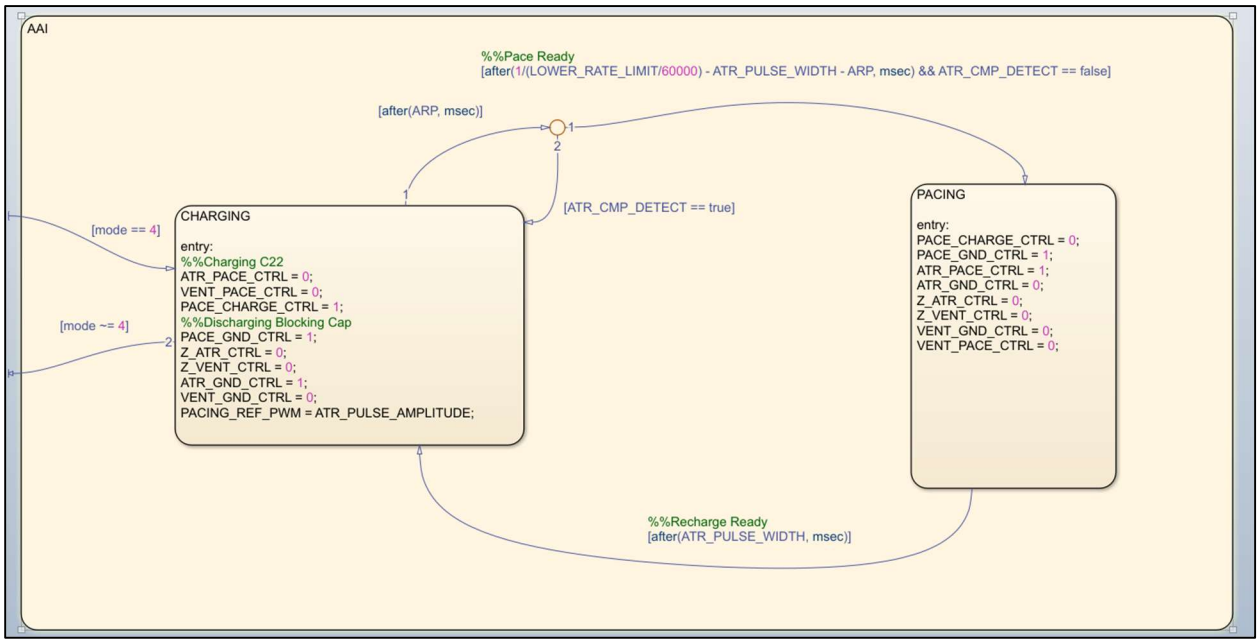


Figure 17: Nested AAI Stateflow

AAI Mode Overview: The AAI mode provides atrial pacing at a rate determined by the lower rate limit, with the added capability to sense natural atrial activity and inhibit pacing when appropriate. This mode outputs pulses at the specified rate only when no intrinsic heartbeat is detected, ensuring pacing support when needed and preventing unnecessary stimulation.

Sensing: In AAI mode, atrial sensing is employed to determine the appropriate timing for pacing. This process relies on the output from the ATR_CMP_DETECT pin within the sensing subsystem. The threshold voltage for comparison is established based on the input from ATR_SENSITIVITY, which is scaled to generate a PWM signal at ATR_CMP_REF_PWM to charge the comparison capacitor.

Charging/Discharging State: The charging/discharging state controls the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined ATR_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, allowing the capacitor to discharge.

Pacing State: The pacing state controls the delivery of the pacing pulse to the atrium. When ATR_PACE_CTRL is set high, capacitor C22 discharges into the atrium, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring that current flows only in the intended direction.

Stateflow Timing: The transition between states from charging to pulsing is governed by the lower rate limit, specified in pulses per minute and converted to pulses per millisecond by dividing by 60,000. The inverse of this value determines the total period. The system switches from the charging state to a junction state after a duration defined by the atrial refractory period (ARP). At this junction, if the ATR_CMP_DETECT input pin reads high, the state reverts to charging, allowing for the ARP duration to elapse. Conversely, if no intrinsic activity is detected, the state transitions to pulsing after a duration equal to the total period minus the pulse width and ARP, returning to charging following the pulse width interval. This is to ensure the paces are delivered at the specified rate. With each state entry, the output controls are adjusted to align with the operational requirements.

2.4.13 AOOR Stateflow

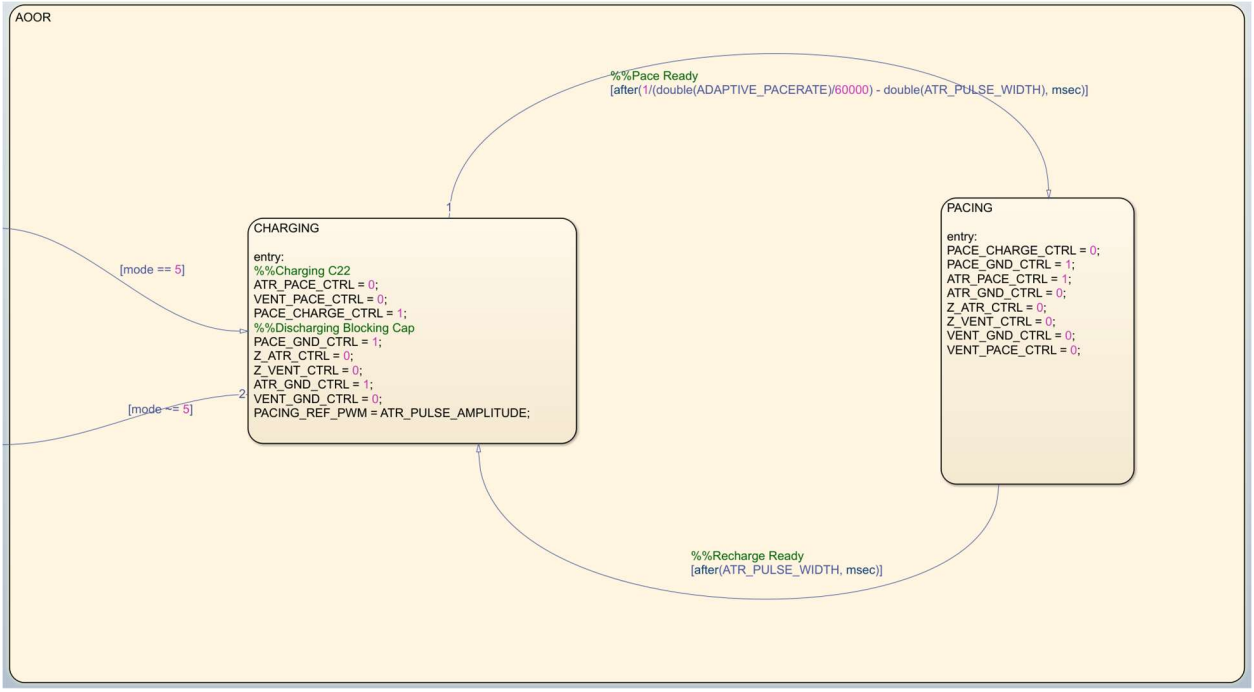


Figure 18: Nested AOOR Stateflow

AOOR Mode Overview: The AOOR mode provides constant, asynchronous pacing to the atrium, with the pacing rate determined by the dynamically updated adaptive pace rate based on the patient's activity level. This ensures that the pacemaker adapts to the patient's physiological needs while delivering consistent stimulation without sensing natural heartbeats.

Charging/Discharging State: The charging/discharging state manages the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined ATR_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, enabling the capacitor to discharge.

Pacing State: The pacing state handles the delivery of the pacing pulse to the atrium. When ATR_PACE_CTRL is set high, capacitor C22 discharges into the atrium, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring unidirectional current flow.

Stateflow Timing: State transitions from charging to pulsing are governed by the dynamically updated lower rate limit. The adaptive pace rate, input in pulses per minute, is divided by 60000

to yield pulses per millisecond, with its value adjusted in real-time based on the patient's activity level. Dividing 1 by this value determines the total period. The state switches from charging to pulsing after a duration equal to the period minus the pulse width and reverts to charging after the pulse width duration. Upon each state entry, the output controls adjust according to the operational requirements and updated rate settings.

2.4.14 VOOR Stateflow

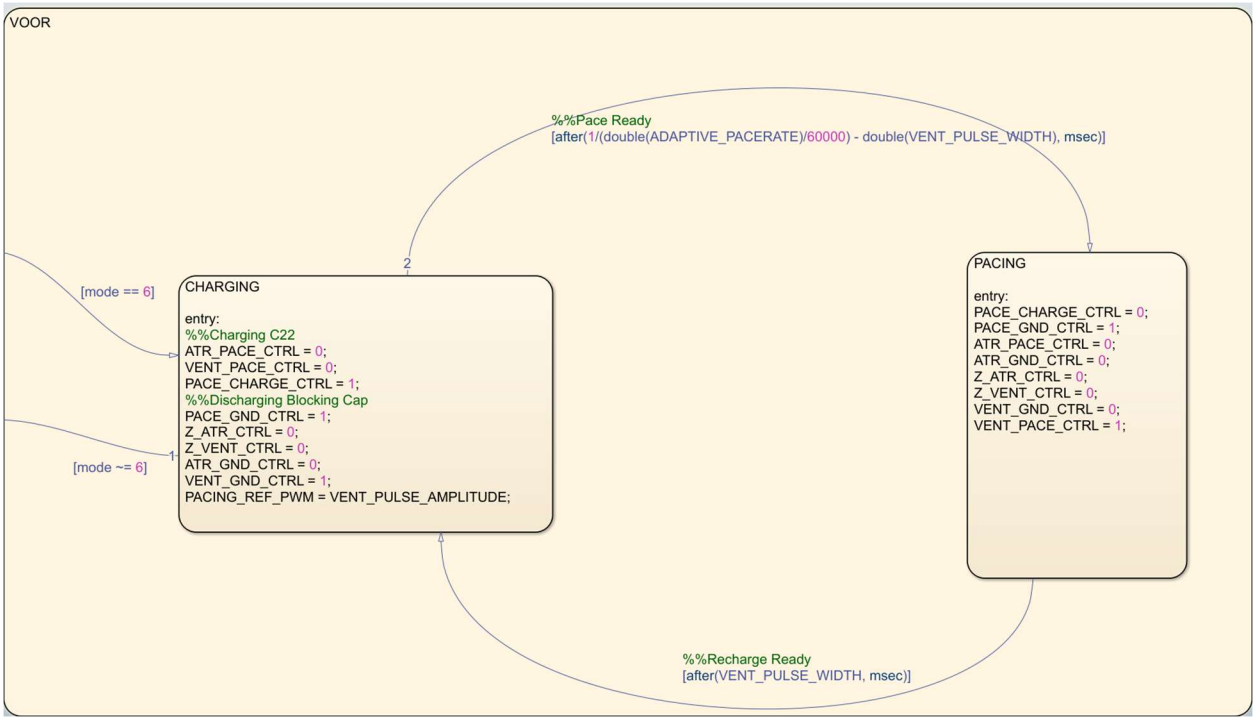


Figure 19: Nested VOOR Stateflow

VOOR Mode Overview: The VOOR mode provides constant, asynchronous pacing to the ventricle, with the pacing rate determined by the dynamically updated adaptive pace rate based on the patient's activity level. This ensures that the pacemaker adapts to the patient's physiological needs while delivering consistent stimulation without sensing natural heartbeats.

Charging/Discharging State: The charging/discharging state manages the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined VENT_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To

discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, enabling the capacitor to discharge.

Pacing State: The pacing state handles the delivery of the pacing pulse to the ventricle. When VENT_PACE_CTRL is set high, capacitor C22 discharges into the ventricle, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring unidirectional current flow.

Stateflow Timing: State transitions from charging to pulsing are governed by the dynamically updated adaptive pace rate. The adaptive pace rate, input in pulses per minute, is divided by 60000 to yield pulses per millisecond, with its value adjusted in real-time based on the patient's activity level. Dividing 1 by this value determines the total period. The state switches from charging to pulsing after a duration equal to the period minus the pulse width and reverts to charging after the pulse width duration. Upon each state entry, the output controls adjust according to the operational requirements and updated rate settings.

2.4.15 AAIR Stateflow

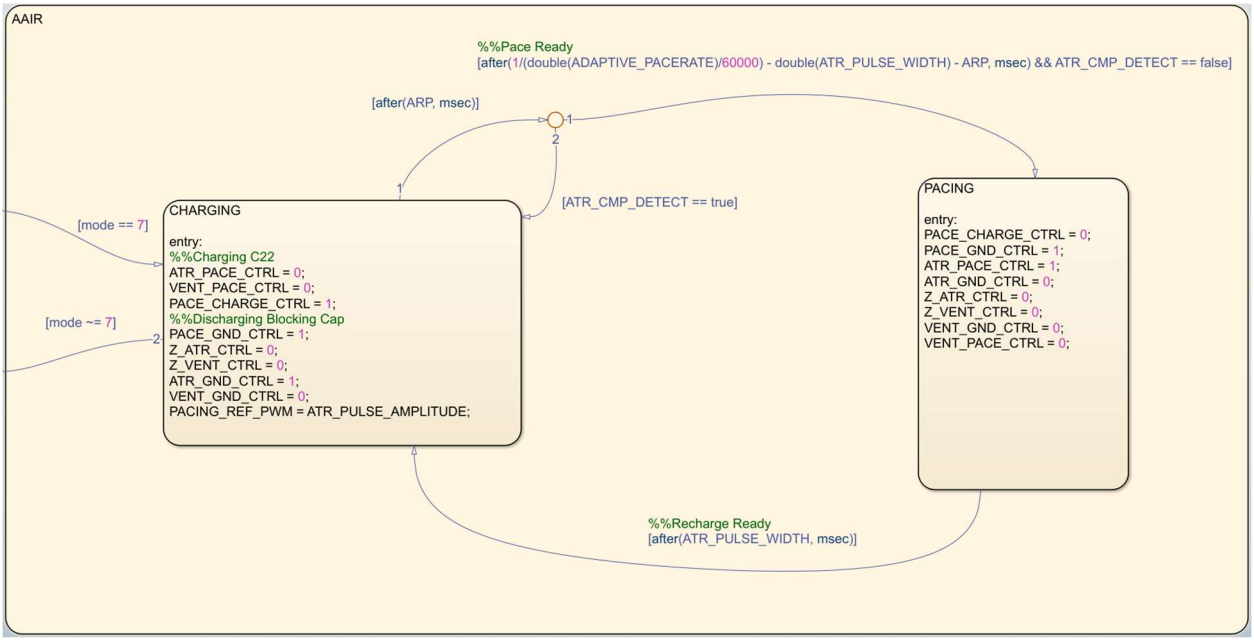


Figure 20: Nested AAIR Stateflow

AAIR Mode Overview: The AAIR mode provides atrial pacing at a rate determined by the dynamically updated adaptive pace rate, with the capability to sense natural atrial activity and

inhibit pacing when appropriate. This mode ensures pacing support only when needed, preventing unnecessary stimulation while adapting to the patient's activity level.

Sensing: In AAIR mode, atrial sensing is employed to determine the appropriate timing for pacing. This process relies on the output from the ATR_CMP_DETECT pin within the sensing subsystem. The threshold voltage for comparison is established based on the input from ATR_SENSITIVITY, which is scaled to generate a PWM signal at ATR_CMP_REF_PWM to charge the comparison capacitor.

Charging/Discharging State: The charging/discharging state manages the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined ATR_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, enabling the capacitor to discharge.

Pacing State: The pacing state handles the delivery of the pacing pulse to the atrium. When ATR_PACE_CTRL is set high, capacitor C22 discharges into the atrium, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring unidirectional current flow.

Stateflow Timing: The transition between states from charging to pulsing is governed by the dynamically updated adaptive pace rate. The adaptive pace rate, specified in pulses per minute, is converted to pulses per millisecond by dividing by 60,000. The inverse of this value determines the total period. The system transitions from the charging state to a junction state after a duration defined by the atrial refractory period (ARP). At this junction, if the ATR_CMP_DETECT input pin reads high, the state reverts to charging, allowing for the ARP duration to elapse. Conversely, if no intrinsic activity is detected, the state transitions to pulsing after a duration equal to the total period minus the pulse width and ARP, returning to charging following the pulse width interval. This ensures paces are delivered in alignment with the adaptive pace rate. With each state entry, the output controls are adjusted to meet the operational requirements.

2.4.16 VVIR Stateflow

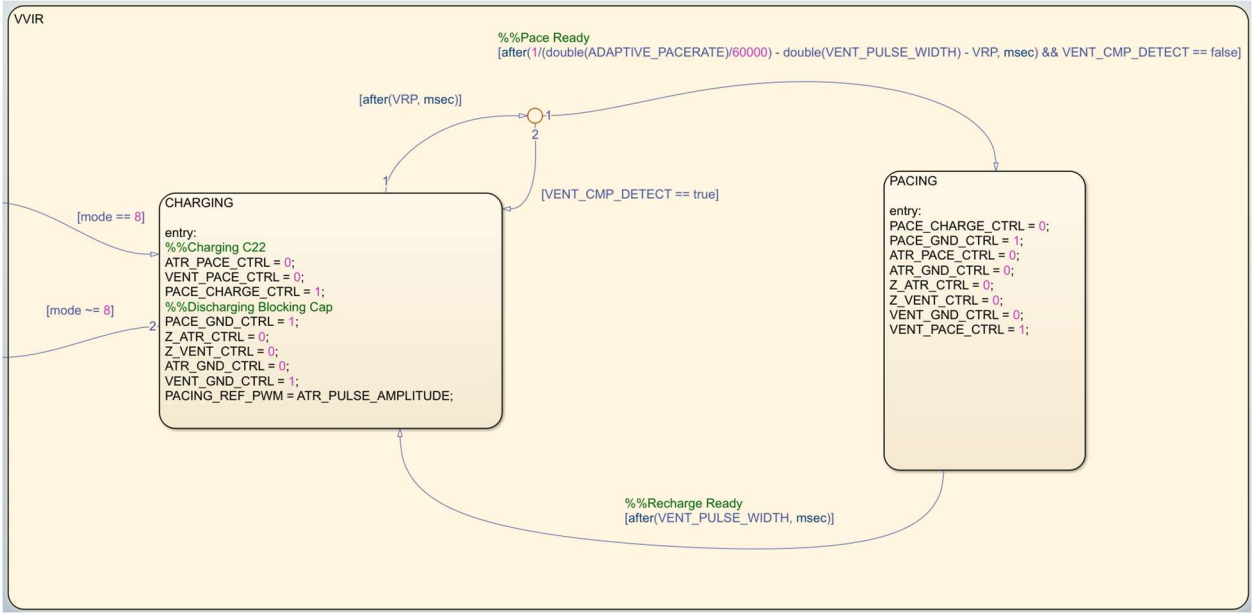


Figure 21: Nested VVIR Stateflow

VVIR Mode Overview: The VVIR mode provides ventricular pacing at a rate determined by the dynamically updated adaptive pace rate, with the added capability to sense natural ventricular activity and inhibit pacing when appropriate. This mode ensures pacing support only when needed while adapting to the patient's activity level, preventing unnecessary stimulation.

Sensing: In VVIR mode, ventricular sensing is employed to determine the appropriate timing for pacing. This process relies on the output from the VENT_CMP_DETECT pin within the sensing subsystem. The threshold voltage for comparison is established based on the input from VENT_SENSITIVITY, which is scaled to generate a PWM signal at VENT_CMP_REF_PWM to charge the comparison capacitor.

Charging/Discharging State: The charging/discharging state manages the charging of capacitor C22, which stores the pacing energy, and the discharging of the blocking capacitor C1. When PACE_CHARGE_CTRL is set to high, current flows into C22. The voltage stored in C22 after charging is determined by the user-defined VENT_PULSE_AMPLITUDE, which is divided by 5 and scaled by 100 before being applied to PACING_REF_PWM to set the charge level. To discharge C21, VENT_GND_CTRL and PACE_GND_CTRL are set to high, enabling the capacitor to discharge.

Pacing State: The pacing state handles the delivery of the pacing pulse to the ventricle. When VENT_PACE_CTRL is set high, capacitor C22 discharges into the ventricle, delivering a pacing pulse. This also charges the blocking capacitor C21, ensuring unidirectional current flow.
Stateflow Timing: The transition between states from charging to pulsing is governed by the dynamically updated adaptive pace rate. The adaptive pace rate, specified in pulses per minute, is converted to pulses per millisecond by dividing by 60,000. The inverse of this value determines the total period. The system transitions from the charging state to a junction state after a duration defined by the ventricular refractory period (VRP). At this junction, if the VENT_CMP_DETECT input pin reads high, the state reverts to charging, allowing for the VRP duration to elapse. Conversely, if no intrinsic activity is detected, the state transitions to pulsing after a duration equal to the total period minus the pulse width and VRP, returning to charging following the pulse width interval. This ensures paces are delivered in alignment with the adaptive pace rate. With each state entry, the output controls are adjusted to align with the operational requirements.

2.5 Testing and Results

Testing was conducted using the provided pacemaker board and the HeartView desktop application to observe pacing signals. The following tests were performed:

- **Mode Switching**: Verified correct pacing behavior for each mode (AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, and VVIR) by observing the pacing signal output.
- **Heart Rate Fluctuation**: Verified correct sensing/pacing behavior for changes in BPM (Under and over the lower rate limit).
- **Activity Fluctuation**: Confirmed correct sensing/pacing behavior for changes in activity detected by the pacemaker.
- **Parameter Adjustments**: Tested the system's response to changes in pulse width, amplitude, and rate limits.
- **Pin Mapping Validation**: Confirmed that pin mapping abstraction worked correctly, with the correct hardware interface mapped without modifying the core model.

Testing was performed through extensive trial and error with the pacemaker and heart view outputs. By constantly viewing different waveform models and comparing results to PACEMAKER documents, we were able to conclude whether our systems met the required specifications of each mode.

Table 2: AOO Test 1

Pacemaker Settings	Heartview Settings
Mode: AOO	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 118 bpm
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker paces the atrium consistently while the simulated heart atrium is active	
Actual Output: There is no AOO pacing showing up at all	
Fix: Wrong pins enabled in the state chart, PACE_CHARGE_CTRL = 0 should be enabled	
PASS/FAIL: FAIL	

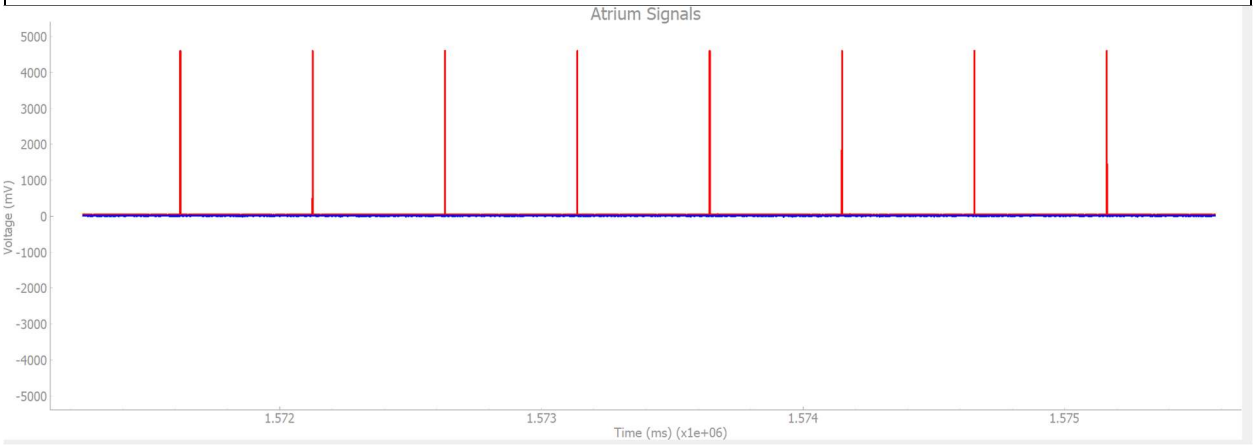


Figure 22: AOO Test 1

Table 3: AOO TEST 2

Pacemaker Settings	Heartview Settings
Mode: AOO	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 77 bpm
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker paces the atrium consistently while the simulated heart atrium is active	
Actual Output: The pacemaker paces the atrium consistently while the simulated heart atrium is active	
PASS/FAIL: PASS	

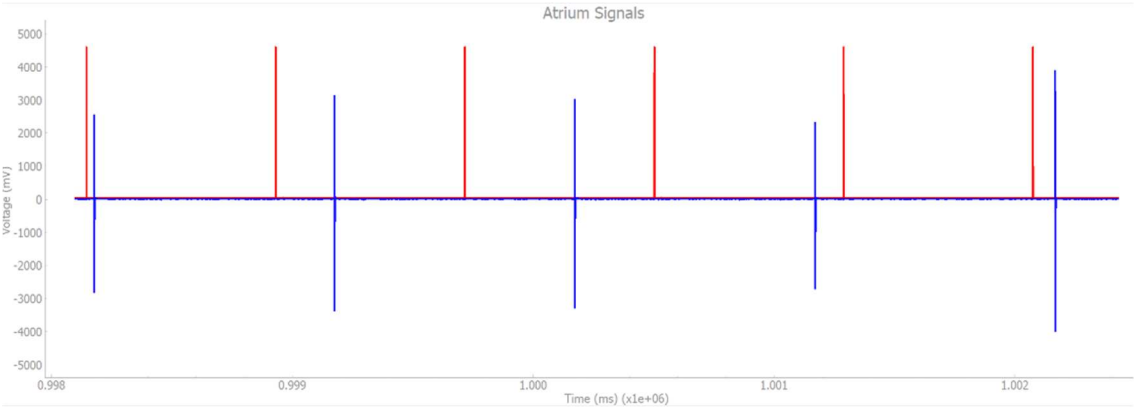


Figure 23: AOO Test 2

Table 4: VOO TEST 1

Pacemaker Settings	Heartview Settings
Mode: VOO	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5V	Natural Heart Rate: 118 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms

Expected Output: The pacemaker paces the ventricle consistently while the simulated heart ventricle is active
Actual Output: Pacing but timing errors cause constant pacing
Fix: Transition state timing was wrong, didn't include width and lower rate
PASS/FAIL: FAIL

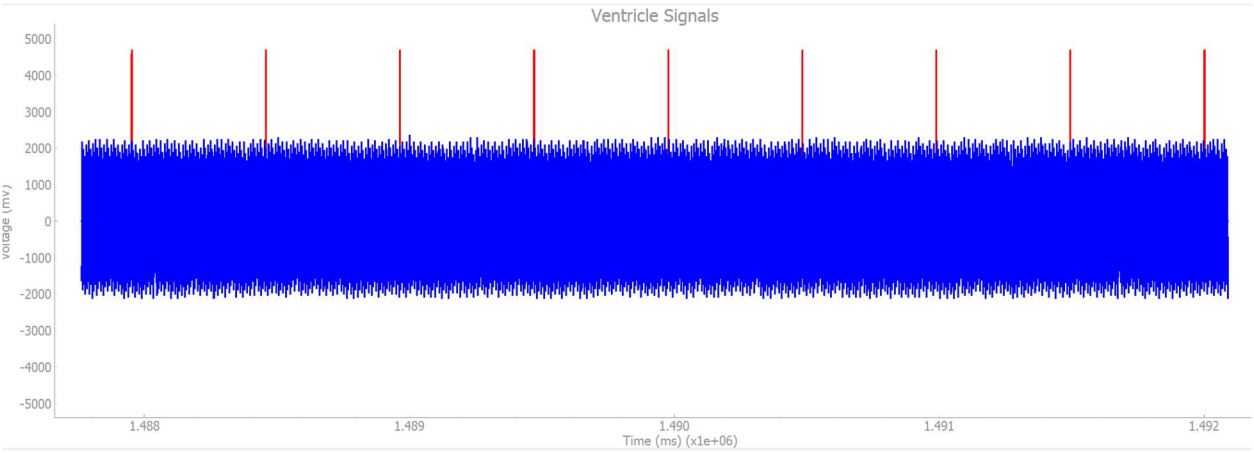


Figure 24: VOO Test 1

Table 5: VOO TEST 2

Pacemaker Settings	Heartview Settings
Mode: VOO	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5V	Natural Heart Rate: 76 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker paces the ventricle consistently while the simulated heart ventricle is active	
Actual Output: The pacemaker paces the ventricle consistently while the simulated heart ventricle is active	
PASS/FAIL: PASS	

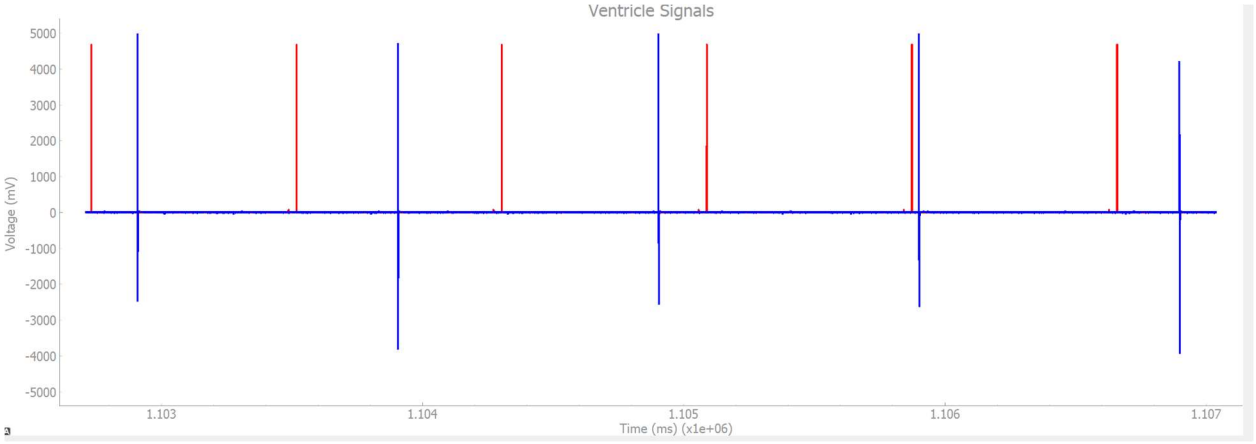


Figure 25: VOO Test 2

Table 6: VVI TEST 1

Pacemaker Settings	Heartview Settings
Mode: VVI	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5V	Natural Heart Rate: 118 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker senses the simulated heart ventricle beating above 60 bpm and does not pace the ventricle	
Actual Output: Doesn't pace but every couple thousand ms, a pace waveform will show	
Fix: Lower Rate Limit and Sensitivity not implemented correctly	
PASS/FAIL: FAIL	

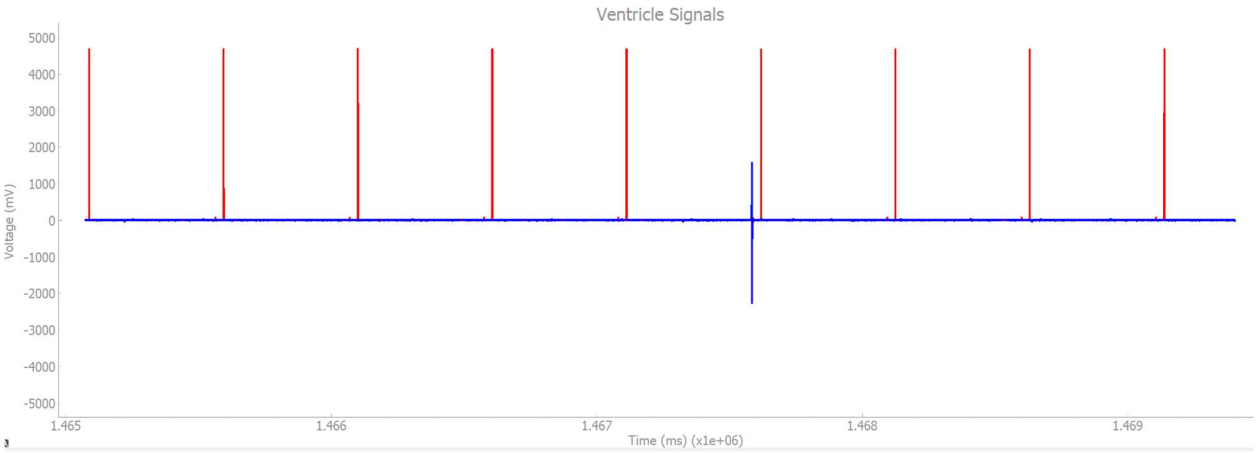


Figure 26: VVI Test 1

Table 7: VVI TEST 2

Pacemaker Settings	Heartview Settings
Mode: VVI	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5V	Natural Heart Rate: 30 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30 ms
Expected Output: The pacemaker paces the ventricle consistently while the simulated heart ventricle is below 60 bpm	
Actual Output: The pacemaker paces the ventricle consistently while the simulated heart ventricle is below 60 bpm	
PASS/FAIL: PASS	

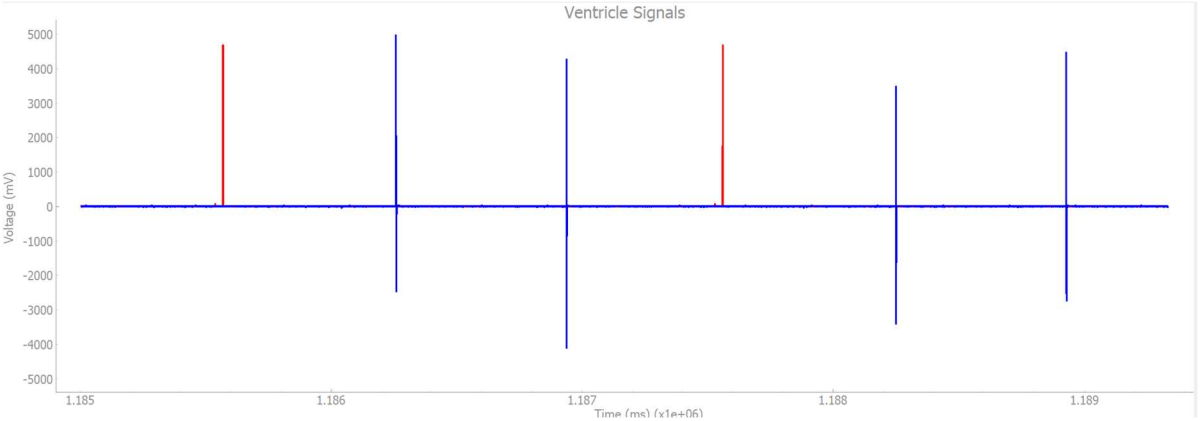


Figure 27: VVI Test 2

Table 8: VVI TEST 3

Pacemaker Settings	Heartview Settings
Mode: VVI	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5 V	Natural Heart Rate: 130 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker senses the simulated heart ventricle beating above 60 bpm and does not pace the ventricle	
Actual Output: The pacemaker senses the simulated heart ventricle beating above 60 bpm and does not pace the ventricle	
PASS/FAIL: PASS	

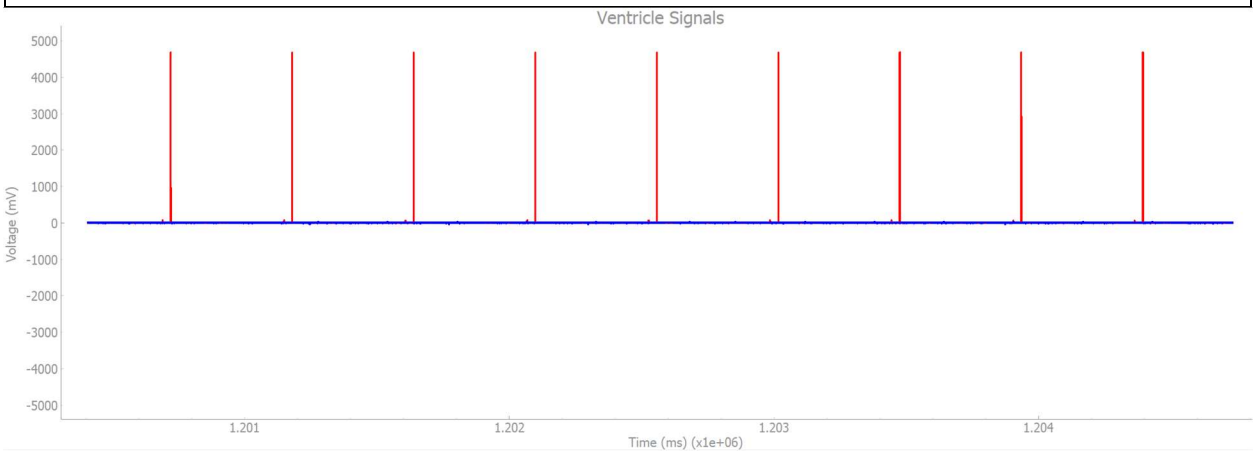


Figure 28: VVI Test 3

Table 9: AAI TEST 1

Pacemaker Settings	Heartview Settings
Mode: AAI	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 142 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker senses the simulated heart atrial beating above 60 bpm and does not pace the atrial	
Actual Output: At 142, the pacemaker still paces even though this heart rate is greater than the lower rate	
Fix: Sensitivity and Lower Rate values entered incorrectly, but theory correct	
PASS/FAIL: FAIL	

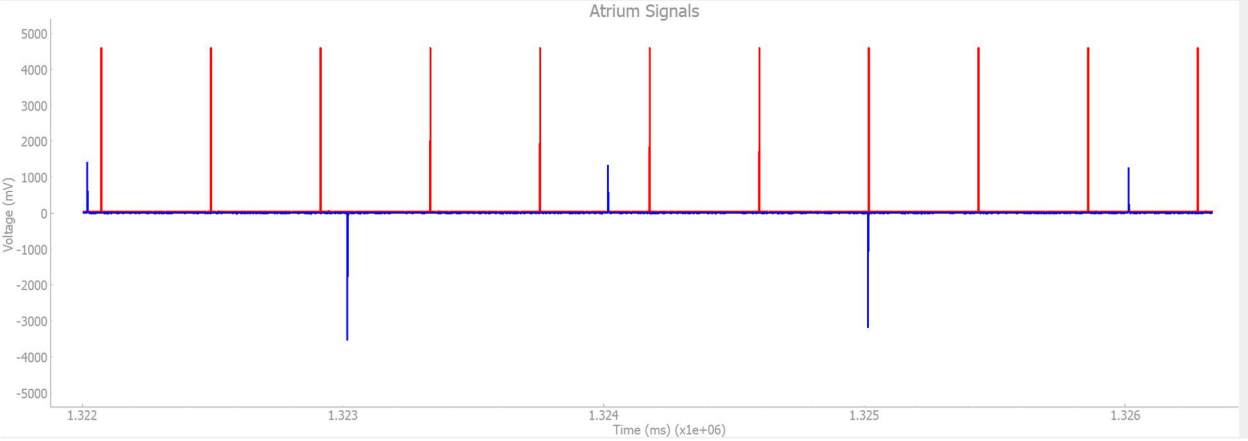


Figure 29: AAI Test 1

Table 10: AAI TEST 2

Pacemaker Settings	Heartview Settings
Mode: AAI	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF

Pulse Amplitude: 3.5V	Natural Heart Rate: 30 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker paces the atrial consistently while the simulated heart atrial is below 60 bpm	
Actual Output: The pacemaker paces the atrial consistently while the simulated heart atrial is below 60 bpm	
PASS/FAIL: PASS	

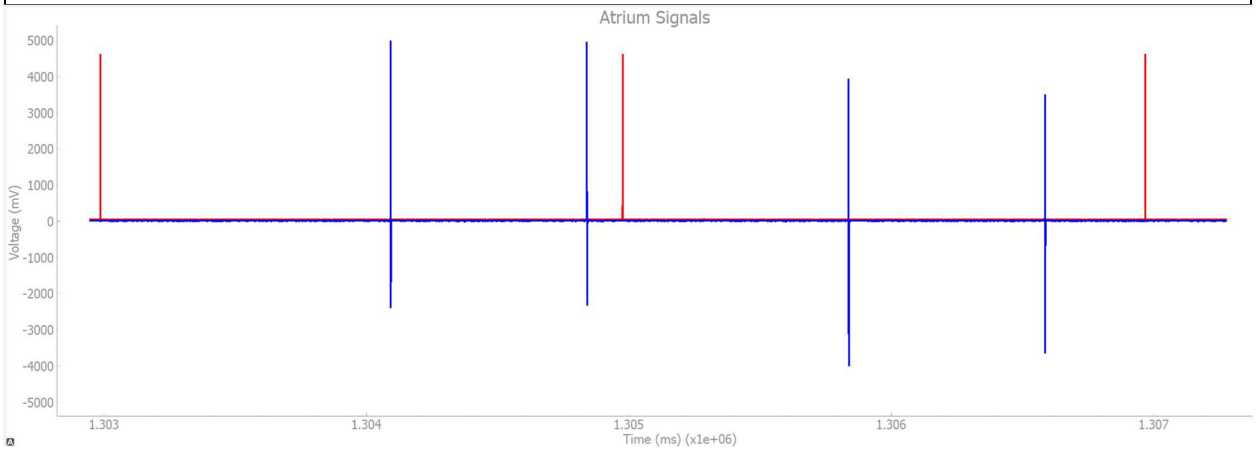


Figure 30: AAI Test 2

Table 11: AAI TEST 3

Pacemaker Settings	Heartview Settings
Mode: AAI	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 60 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker senses the simulated heart atrial beating above 60 bpm and does not pace the atrial	
Actual Output: The pacemaker senses the simulated heart atrial beating above 60 bpm and does not pace the atrial	
PASS/FAIL: PASS	

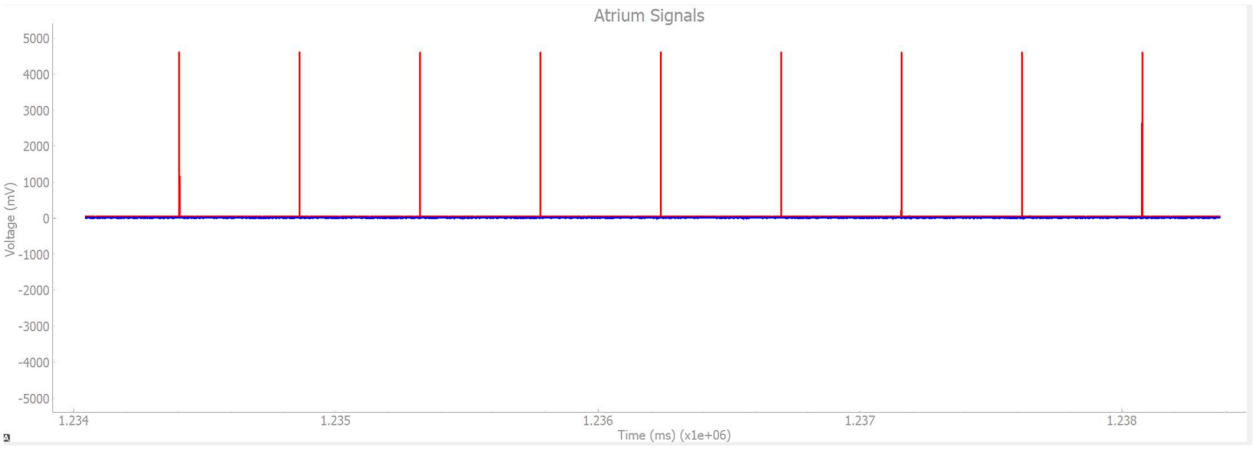


Figure 31: AAI Test 3

Table 12 12: AOOR TEST 1

Pacemaker Settings	Heartview Settings
Mode: AOOR	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 60 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker should consistently pace and when the device is shaken, it should pace at a faster rate	
Actual Output: The pacing is going at a slower rate, with heart view showing wider heart beats.	
Fix: The code had subtractions instead of additions which provided the wrong logic	
PASS/FAIL: FAIL	

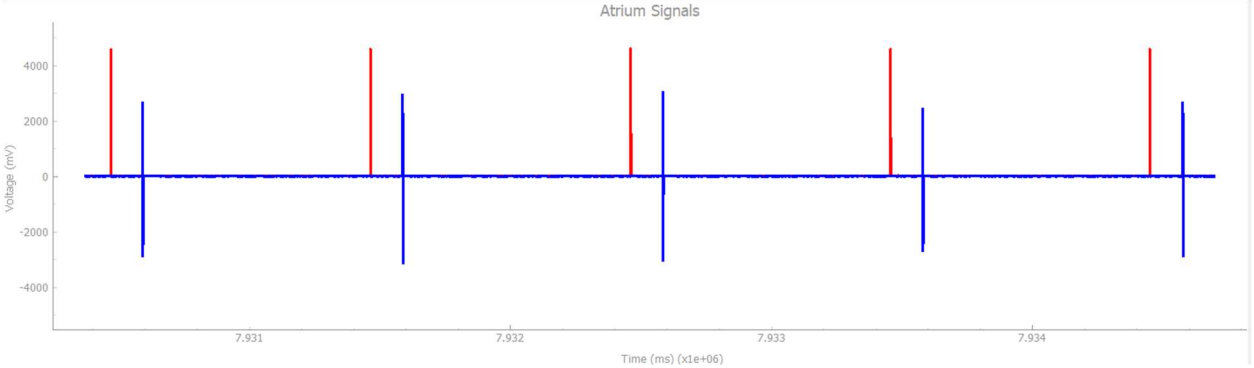


Figure 32: AOOR Pacing before shaken

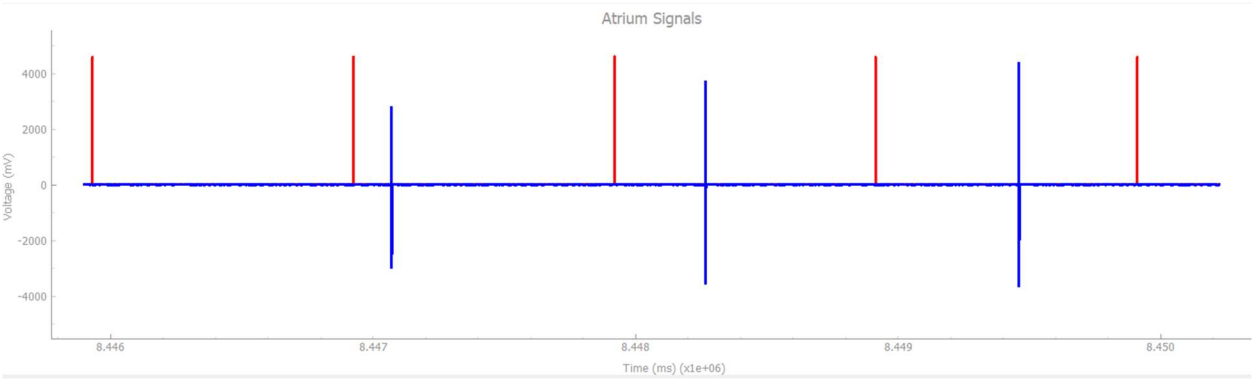


Figure 33: AOOR Pacing does not change after device is shaken

Table 13 13: AOOR TEST 2

Pacemaker Settings	Heartview Settings
Mode: AOOR	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 60 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker should consistently pace and when the device is shaken, it should pace at a faster rate	
Actual Output: As the device is shaken, the heart paces at a faster rate, shrinking the widths of the atrium signals	
PASS/FAIL: PASS	

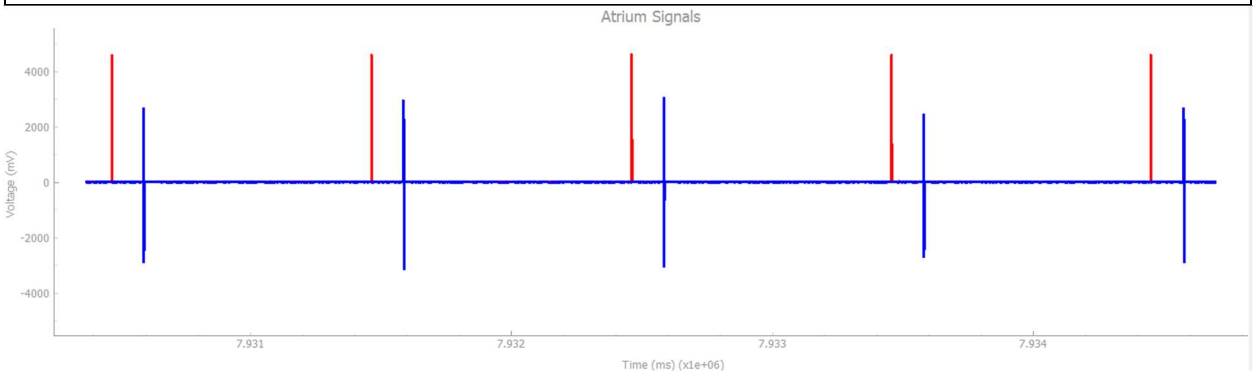


Figure 34: AOOR Pacing before the device is shaken.

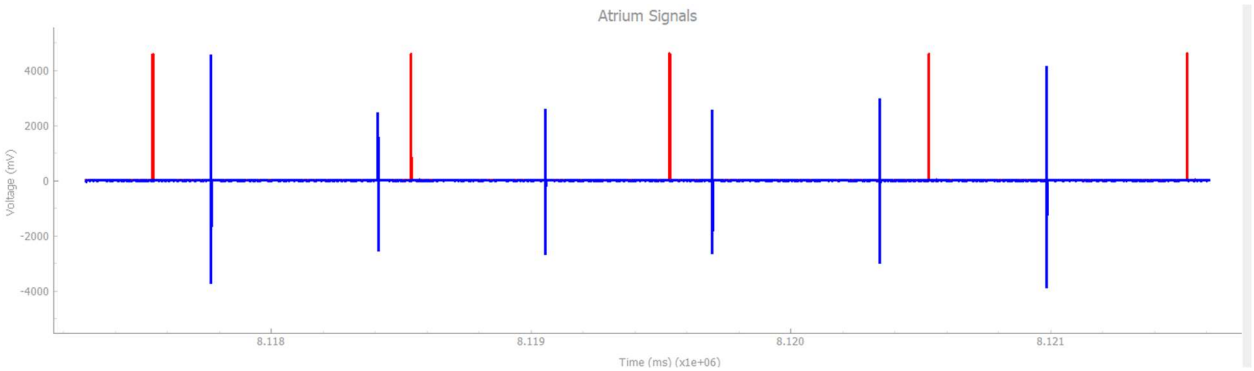


Figure 35: AOOR Pacing speed increases after the device is shaken.

Table 14 14: VOOR TEST 1

Pacemaker Settings	Heartview Settings
Mode: VOOR	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5V	Natural Heart Rate: 60 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker should consistently pace the ventricle and when the device is shaken, it should pace at a faster rate	
Actual Output: Nothing changes, the ventricle signal paces at a similar rate	
FIX: The moving average buffer was too high, stopping any data from being shown on heartview and preventing anything from being updated.	
PASS/FAIL: FAIL	

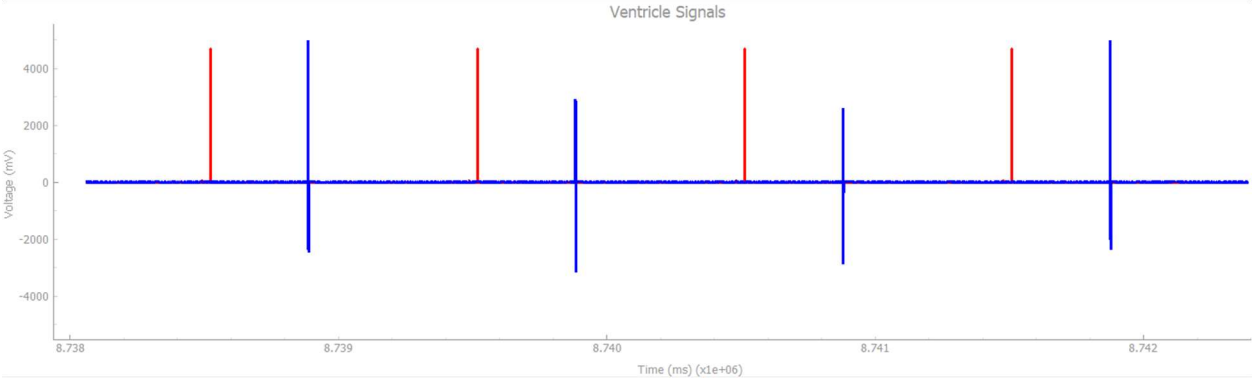


Figure 36: VOOR Ventricle signal pacing before being shaken

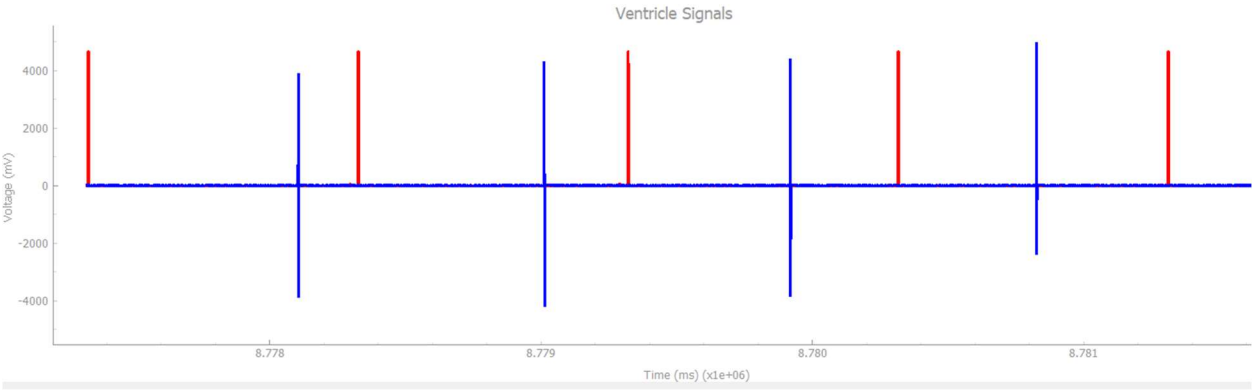


Figure 37: VOOR Ventricle signal pacing after being shaken

Table 1515: VOOR TEST 2

Pacemaker Settings	Heartview Settings
Mode: VOOR	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 3.5V	Natural Heart Rate: 60 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: The pacemaker should consistently pace the ventricle and when the device is shaken, it should pace at a faster rate	
Actual Output: As the device is shaken, the heart paces at a faster rate, shrinking the widths of the ventricle signals	
PASS/FAIL: PASS	

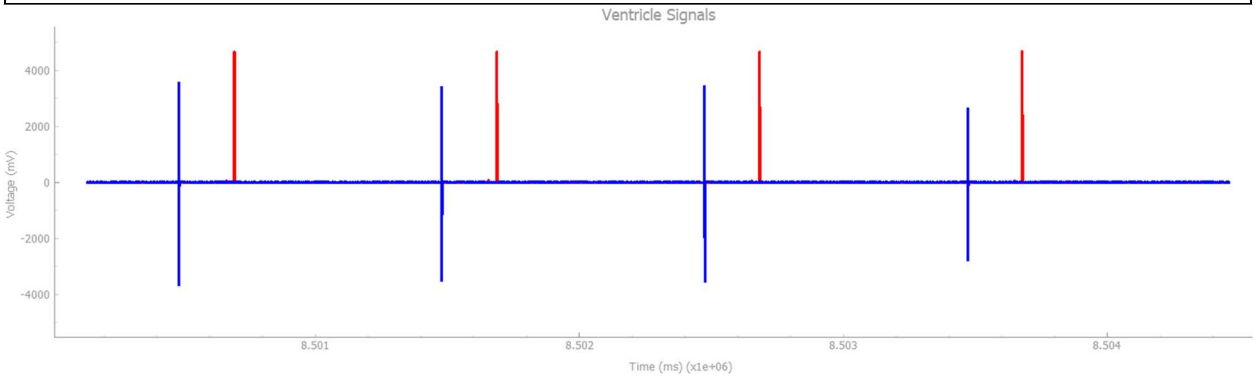


Figure 38: VOOR Ventricle signal pacing before being shaken

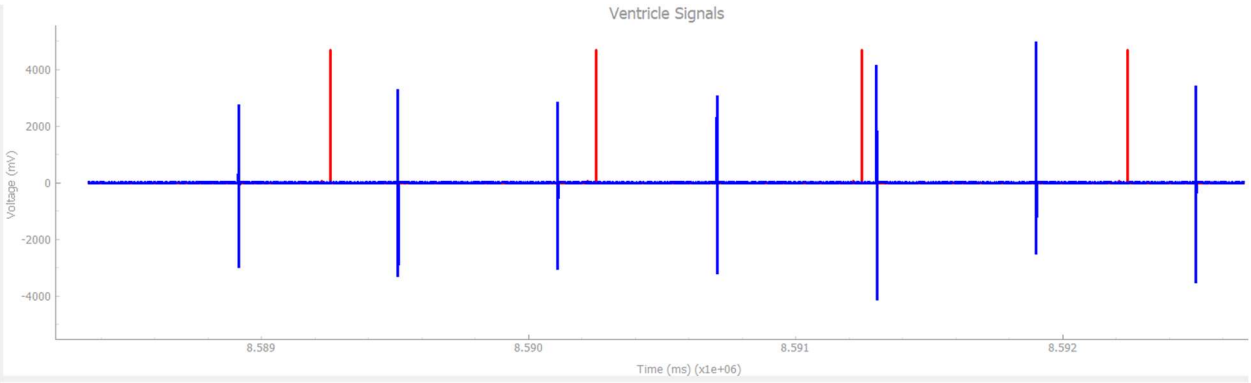


Figure 39: VOOR Ventricle signal pacing after being shaken, pacing rate increases

Table 16 16: AAIR TEST 1

Pacemaker Settings	Heartview Settings
Mode: AAIR	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 80 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: When no activity is detected and the heart rate is above 60 ppm, the device should NOT pace, but when the device detects activity, the device should pace above 60 ppm	
Actual Output: Device doesn't pace with heart above 60 ppm with and without activity detected.	
FIX: Atrium pacing not updated in sync with the adapted pace rate.	
PASS/FAIL: FAIL	

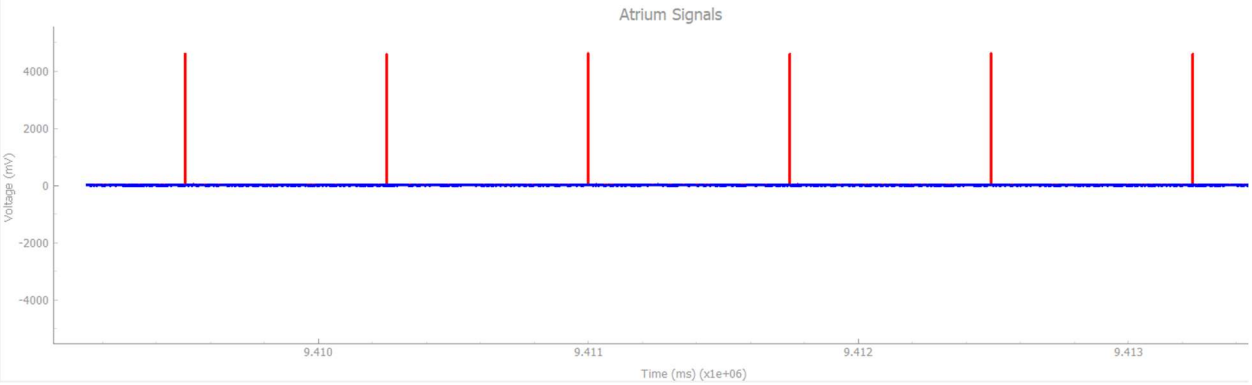


Figure 40: AAIR Pacemaker senses appropriate heart rate and does NOT pace

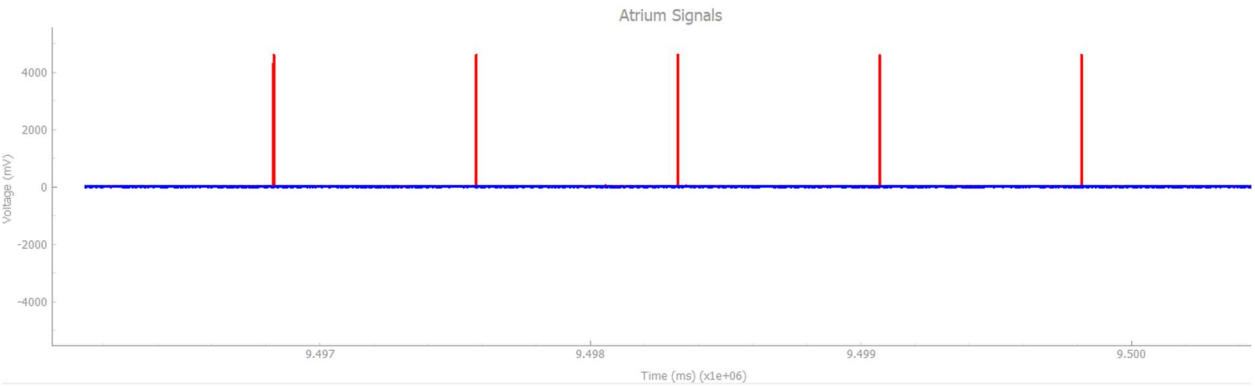


Figure 41: AAIR Pacing after activity detected

Table 1717: AAIR TEST 2

Pacemaker Settings	Heartview Settings
Mode: AAIR	Natural Atrium: ON
Pulse Width: 1ms	Natural Ventricle: OFF
Pulse Amplitude: 3.5V	Natural Heart Rate: 75 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: When no activity is detected and the heart rate is above 60 ppm, the device should NOT pace, but when the device detects activity, the device SHOULD pace above 60 ppm	
Actual Output: Device paces above 60 ppm with activity detected, doesn't pace with heart above 60 ppm without activity detected.	
PASS/FAIL: PASS	

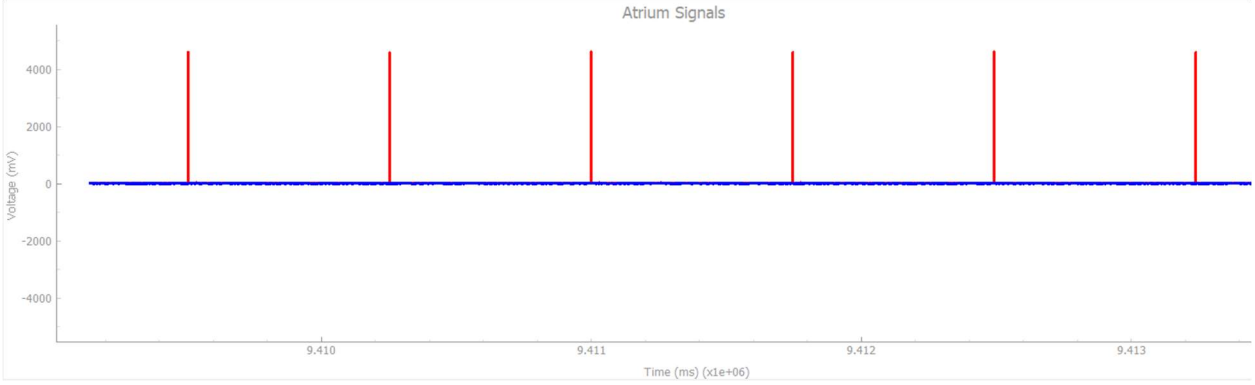


Figure 42: AAIR Atrium signals when heart rate is below 60 ppm

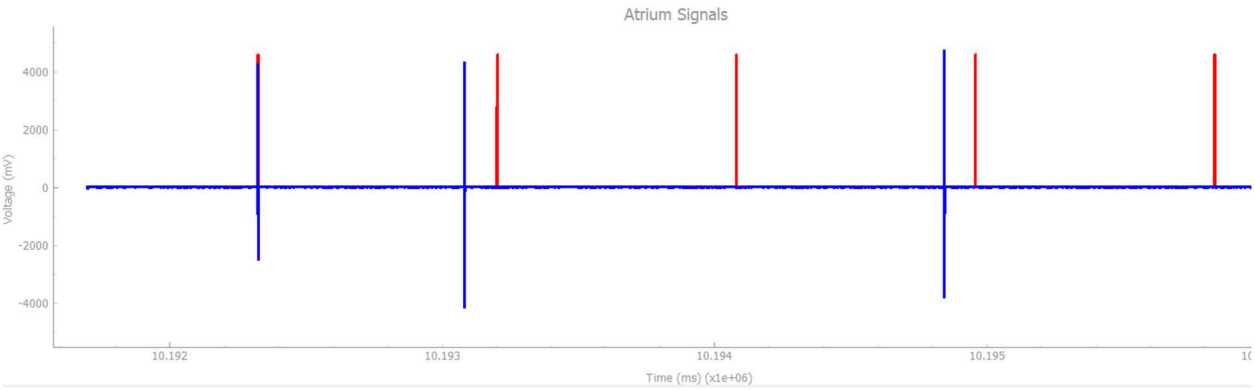


Figure 43: AAIR Paces when activity is detected

Table 19 18: VVIR TEST

Pacemaker Settings	Heartview Settings
Mode: VVIR	Natural Atrium: OFF
Pulse Width: 1ms	Natural Ventricle: ON
Pulse Amplitude: 5 V	Natural Heart Rate: 77 BPM
Lower Rate: 60 ppm	Natural AV Delay: 30ms
Expected Output: Pulses with 5 V of amplitude as the pacemaker detects activity.	
Actual Output: Pulses with 5 V of amplitude as the pacemaker detects activity.	
PASS/FAIL: PASS	

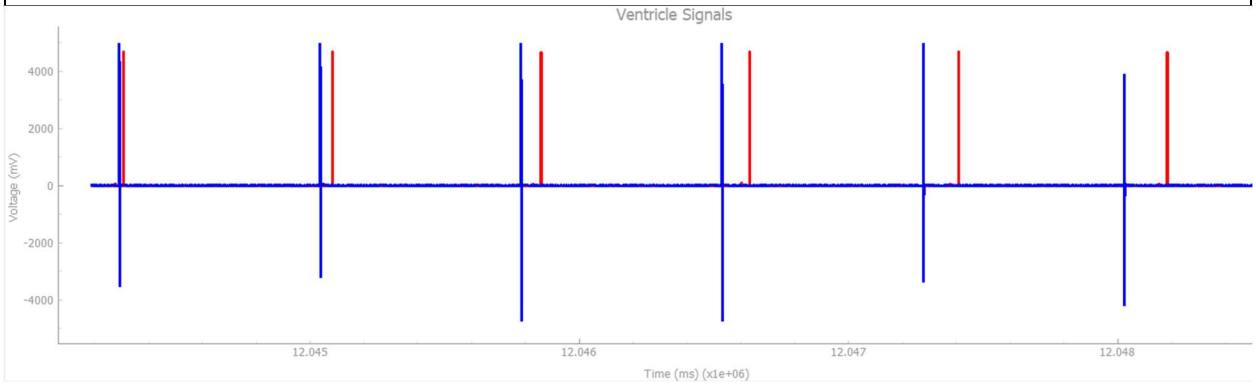


Figure 44: VVIR Working ventricle signals

2.6 Potential Changes and Future Considerations

2.6.1 Requirement Changes (ASSIGNMENT 1)

- Additional support will be added for AOOR, VOOR, AAIR, VVIR modes.
- Parameters will be added for the additional modes' rate modulation functionality.
- Possible changes to the pacing rate limits or additional parameters may be introduced in future versions.
- The system will have to establish communication with and accept input values from the DCM and apply them to the simulation model

Table 20 19: Updated parameters required for each mode

Parameter	AOOR	AAIR	VOOR	VVIR
Lower Rate Limit	X	X	X	X
Upper Rate Limit	X	X	X	X
Maximum Sensor Rate	X	X	X	X
Atrial Amplitude	X	X		
Ventricular Amplitude			X	X
Atrial Pulse Width	X	X		
Ventricular Pulse Width			X	X
Atrial Sensitivity		X		

Ventricular Sensitivity				X
VRP				X
ARP		X		
PVARP		X		
Hysteresis		X		X
Rate Smoothing		X		X
Activity Threshold	X	X	X	X
Reaction Time	X	X	X	X
Response Factor	X	X	X	X
Recovery Time	X	X	X	X

2.6.2 Design Decision Changes (ASSIGNMENT 1)

- The hardware abstraction model may evolve as more complex hardware is introduced, necessitating updates to the subsystem.
- Introduce additional safety requirements, such as a failsafe mode or error detection for out-of-range inputs, to ensure that pacing parameters do not exceed safe limits.
- May add additional states or sub-states in the state flow to handle safety conditions, possibly including real-time monitoring of signal integrity.
- Add a more intuitive way to transition between modes.
- Expand the Stateflow to accommodate for additional modes.

2.6.3 Requirement Changes (POST ASSIGNMENT 2)

- Support will be added for DDD and DDDR modes, including dual-chamber pacing and rate modulation functionality, requiring implementation of parameters such as atrioventricular delay and mode-specific rate settings.

- Enhanced logic will be developed to manage seamless transitions between pacing modes, ensuring consistent operation and interaction between atrial and ventricular sensing and pacing.
- Communication with the DCM will be expanded to handle additional parameters for the new modes, ensuring accurate input processing and feedback within the simulation model.

2.6.4 Design Decision Changes (POST ASSIGNMENT 2)

- Increase modularity and hardware hiding in our main Simulink model by decreasing the number of connections outside of subsystems. This may be done by using more ports, increasing our hardware hiding and protecting data.
- Encrypt UART serial channels to prevent unwanted users from reading protected data.

2.7 Simulink Development History

Table 2120: Simulink Development History

Date	Changes
September 26th, 2024	Created initial Stateflows for VOO, AOO modes. Gathered parameters for each mode from documentation.
October 3rd, 2024	Faced issues with VOO and AOO initial Stateflows. Switched PACING_REF_PWM to a PWM Output, as it was a Digital Output before which caused undesired functionality. Tested AOO and VOO, verified correct functionality of each function.
October 4th, 2024	Created initial Stateflows for VVI, AAI with incorrect timing and sensing. Gathered parameters for VVI and AAI from documentation.
October 10th, 2024	Created an initial combined model to implement hardware hiding. Included a programmable parameters subsystem which was inputted to the main pacemaker Stateflow. Also created a microcontroller pin

	subsystem that takes the output values programmed from the Stateflow as input. Set FRONTEND_CTRL to HIGH in the programmable parameters subsystem. FRONTEND_CTRL wasn't included previously, resulting in incorrect sensing.
October 16th, 2024	Corrected timing for VVI, AAI by utilizing the VRP/ARP in the transition to pacing. Confirmed correctness of VVI and AAI models.
October 22nd, 2024	Removed sensing parameters from programmable parameters subsystem, added them to a separate sensing subsystem. Added AOO, VOO, AAI, VVI Stateflows to the combined model. Confirmed correctness of each mode in the combined model by switching the mode value.
November 7th, 2024	Gathered and discussed future plans for project and how we will be addressing new requirements Worked on Tutorial 4, trying to create a python script for UART communication between device and Simulink, changing LED colour.
November 14th, 2024	Began testing rate adaptive pacing modes. Wrote Pacing rate formulas, calculating moving averages, and outputting a pacing rate to the main Stateflow Added AOOR, VOOR, AAIR, VVIR modes to the main Stateflow model. Confirmed correctness of these new modes
November 21st, 2024	Worked on UART communication with both DCM and Simulink, figuring out how to pack and unpack data

	Figured out how to trigger global functions and receive data from python script
November 27th, 2024	Finalized adding new modes to main Stateflow model. Tested and confirmed correctness of switching between modes and having default mode that does nothing in the pacemaker. Finalized byte packing and unpacking data.

3. The Device Controller-Monitor (DCM)

3.1 The DCM

The Device Controller-Monitor (DCM) is a user application used to manage and interact with the pacemaker model operating on an FRDM-K64F microcontroller. Through a graphical user interface (GUI), the DCM provides users with an easy way to adjust the pacemaker's programmable settings and select the modes available to be run on the Pacemaker. This interface includes interactive elements such as sliders for adjusting parameter values, drop-downs for mode selections, and other accessibility features to enhance user experience. In assignment 2, the DCM's Communication Layer will translate user inputs to the pacemaker, establishing a reliable two-way link with the microcontroller. This setup allows for simple control of the pacemaker's parameters and modes from a user-friendly desktop application.

3.2 Requirements

This section outlines the requirements for developing the Device Controller-Monitor (DCM) interface. The DCM should be user-friendly, contain user security features and provide the necessary inputs for managing pacemaker parameters and modes. The DCM must contain the following elements ensure these requirements are met.

3.2.1 Feature Requirements

1. Welcome Screen:
    - The welcome screen must enable the user with the ability to log in, (if the user has previously registered on the platform) or register if they are a new user.
    - The Welcome screen must allow a maximum of 10 users to register.

2. User interface:
    - The interface shall support managing windows display both text and graphical content.
    - The interface must allow the user to interact using input buttons and process their positioning.
    - The interface will display all programmable parameters for easy review and modification.
    - The DCM will include a visual indicator to show when communication between the DCM and the pacemaker device is active.

3. Device communication status:
    - The DCM must indicate visually when the DCM is actively communicating with the pacemaker device.

3.2.2 Parameter and Mode Input Requirements

1. The DCM must provide the user with an option to select one of the following pacemaker modes:
    - VOO
    - AOO
    - VVI
    - AAI
    - AOOR
    - VOOR

- AAIR
- VVIR

2. The DCM interface will feature sliders for adjusting the pacemaker's programmable parameters. Each slider will have a specified range, with a defined starting value (left), an upper limit (middle), and increments (right) to ensure precise adjustments. Below is a breakdown of the sliders and their configurations:

    - Lower Rate Limit (ppm): Range from 30 to 175, adjustable in increments of 5.
    - Upper Rate Limit (ppm): Range from 50 to 175, adjustable in increments of 5.
    - Maximum Sensor Rate (ppm): Range from 50 to 175, adjustable in increments of 5.
    - Atrial Amplitude (V): Range from 0.1 to 5.0, adjustable in increments of 0.1.
    - Ventricular Amplitude (V): Range from 0.1 to 5.0, adjustable in increments of 0.1.
    - Atrial Pulse Width (ms): Range from 1 to 30, adjustable in increments of 1.
    - Ventricular Pulse Width (ms): Range from 1 to 30, adjustable in increments of 1.
    - Atrial Sensitivity (mV): Range from 0 to 5, adjustable in increments of 0.1.
    - Ventricular Sensitivity (mV): Range from 0 to 5, adjustable in increments of 0.1.
    - VRP (Ventricular Refractory Period) (ms): Range from 150 to 500, adjustable in increments of 10.
    - ARP (Atrial Refractory Period) (ms): Range from 150 to 500, adjustable in increments of 10.
    - PVARP (Post-Ventricular Atrial Refractory Period) (ms): Range from 150 to 500, adjustable in increments of 10.
    - Rate Smoothing (%): Range from 3 to 25, adjustable in increments of 3.

- Reaction Time (s): Range from 10 to 50, adjustable in increments of 10.
- Response Factor: Range from 1 to 16, adjustable in increments of 1.
- Recovery Time (min): Range from 2 to 16, adjustable in increments of 1.
- Activity Threshold: Range from 0 to 6, adjustable in increments of 1.
- Hysteresis: Range from 0 to 1, adjustable in increments of 1.

3. **Parameter Storage**:
    - The DCM must store all parameters listed above, including the selected mode, to allow for future use.
    - A history of parameter configurations must be maintained, enabling users to review or revert to previous settings.

4. **Parameter Validation and Verification**:
    - The DCM must validate parameter values entered by the user to ensure they meet the specified ranges, increments, and tolerances.
    - The system must confirm that parameters are stored correctly in the Pacemaker device by verifying them after transmission.

5. **Parameter Transmission**:
    - The DCM must send all programmable parameters and the selected mode to the Pacemaker over a serial communication link upon user request.
6. **Egram Data Management**:
    - The DCM must allow users to request and display Egram data for either the atrium, ventricle, or both.
    - The DCM must be able to stop streaming Egram data upon user request.
    - Egram data must be printable if requested by the user.

7. **Interface Enhancements**:
    - The interface must provide mode-specific parameter input fields, ensuring clarity and ease of use.

- Visual indicators must highlight active mode and current device communication status.

3.3 Design Decisions

3.3.1 GUI Language Selection

A graphical user interface (GUI) was chosen as the method for constructing the DCM to provide an intuitive and interactive way for users to manage the pacemaker settings.

Python was selected as the programming language due to its simplicity and the availability of built-in libraries, such as Tkinter, which would assist in streamlining the development process of the graphical user interface. Python's simplicity and ease of use make it an ideal choice for accomplishing the principal goal of creating a user-friendly environment, while still being powerful enough to meet the project's functional requirements.

```
import tkinter as tk
from tkinter import messagebox
from tkinter import ttk
import json
import tkinter.font as tkFont
import bcrypt
```

Figure 45: Python Libraries

Figure 10 provides the code used to import all the Python libraries that were used to create the DCM for assignment one. These libraries were chosen for the GUI's construction because they each offer unique features that contribute to a user-friendly and secure interface. By combining their individual capabilities, we're able to efficiently build a responsive and visually appealing GUI with secure data management.

The **Tkinter** library, Python's standard library for creating GUIs, allows for rapid development of graphical elements, making it well-suited for building interactive user interfaces. Tkinter's **messagebox** module enables the easy creation of pop-up alerts and dialogues, which are helpful for improving the user experience by providing feedback and notifications. This was

used multiple times in the assignment when notifying the user when their sign-in request was successful, when their registration was successful when their parameters were saved and others. Additionally, **ttk,** which is a module within tkinter, supports advanced styling for elements and widgets, which gives the interface a more modern and polished look and feel.

To handle user information and parameters, **json** was selected due to its simplicity in managing data and compatibility with text-based data storing. **Json** allowed for easy retrieval and saving of parameters and user data, which is crucial for managing user profiles and configuration preferences straightforwardly.

The **tkFont** module was chosen to allow customization of font styles and sizes, an important feature for creating a clear, readable interface that is enhanced visually. The ability to adjust font appearance and size ensures that users can easily navigate the DCM's interface.

Finally, **bcrypt** is used for securely storing user passwords. This library applies robust hashing algorithms to protect user credentials, providing an additional layer of security to the DCM application. By integrating bcrypt, we ensure that sensitive information, such as passwords, is securely managed, reducing the risk of unauthorized access.

3.3.2 Welcome Page UI

The welcome page was designed to meet all the requirements of the page located in section 3.2.1.

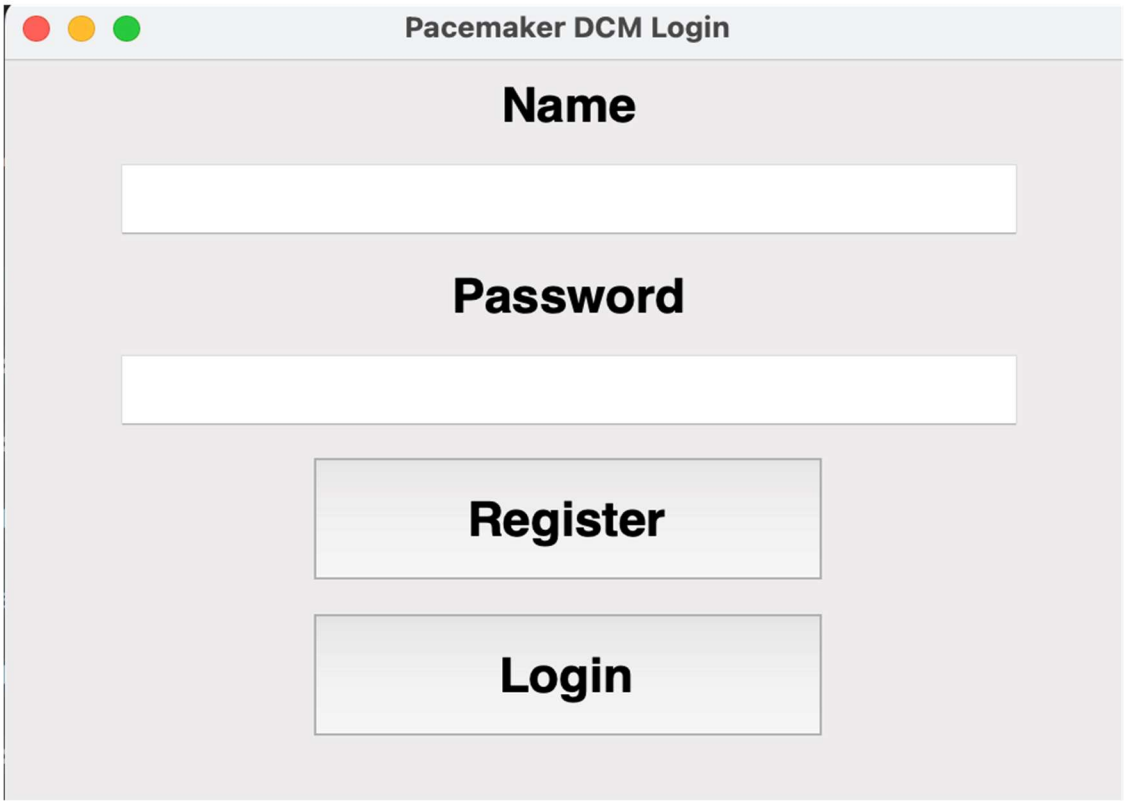


Figure 46: Welcome Page

The welcome page is shown above which includes the two features to either sign in or login.

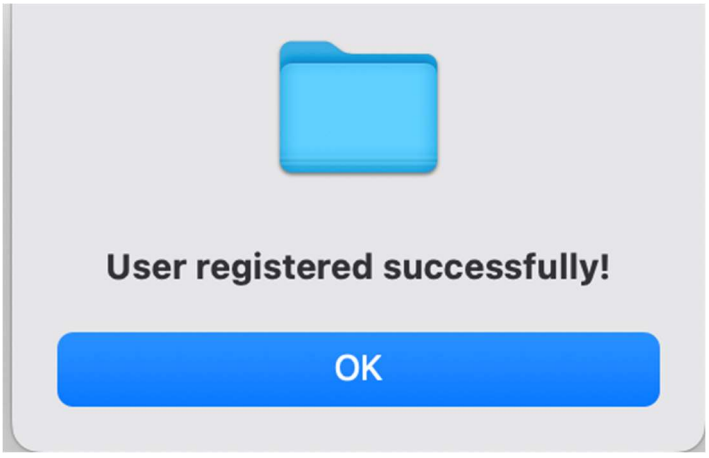


Figure 47: Registration Confirmation

When the user enters a new username which has not been registered by another user and a password of at least 1 character, their new profile will be registered within the Json data file. This will only be succefull when less then 10 users have been registered, as this variable holds a maximum value of 10.

Figure 48: Welcome confirmation

Figure 13 displays the pop-up confirmation after a user logs into the application. This is applicable when a user has previously registered and has entered their correct username and password.

```
def load_users(self):
    try:
        with open(USERS_FILE, 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return {}

def save_users(self):
    with open(USERS_FILE, 'w') as f:
        json.dump(self.users, f, indent=4)
```

Figure 49: Login/Register functions

The above shows the code used to program the load and save users functions. The users and stored in a Json file.

Figure 50: User Data Json File

The Json file above is used to store users' data, including the programmed parameters and the selected mode. These parameters are repopulated every time the user logs in and can be changed using the save button on the main page. The password is hashed using Bycrypt, using the password function below.

```
def hash_password(self, password):
    salt = bcrypt.gensalt()
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')

def check_password(self, hashed_password, password):
    return bcrypt.checkpw(password.encode('utf-8'), hashed_password.encode('utf-8'))
```

Figure 51: Password Encryption Functions

```
def register_user(self):
    name = self.entry_name.get()
    password = self.entry_password.get()

    if name in self.users:
        messagebox.showerror("Error", "User already exists!") #checking if the username has already been registed
        return

    if len(self.users) >= MAX_USERS:
        messagebox.showerror("Error", "Maximum user limit reached!") #checking if their is space for a user
        return

    if name and password:
        self.users[name] = {
            'password': self.hash_password(password),  #if both are valid, we peform the folowing
            'mode': 'AOO'
        }
        self.save_users()
        messagebox.showinfo("Success", "User registered successfully!") #appending the new user to the Json
        self.clear_entries() #clearing the registration fields
    else:
        messagebox.showerror("Error", "Please enter both name and password.") #otherwise error

def login_user(self):
    name = self.entry_name.get()
    password = self.entry_password.get()

    if name in self.users and self.check_password(self.users[name]['password'], password):#using the unhashing function
        messagebox.showinfo("Success", f"Welcome {name}!")                                 #the inputted passcode
        self.open_pacemaker_interface(name) #if valid open
    else:
        messagebox.showerror("Error", "Failed Login Attempt.") #otherwise error
```

Figure 52: User Login/Registration Functions

Both the login and register functions utilize the hash and check password functions to encode and decrypt the user's password to promote privacy and security, one of the requirements of the DCM.

3.3.3 Parameter Interface

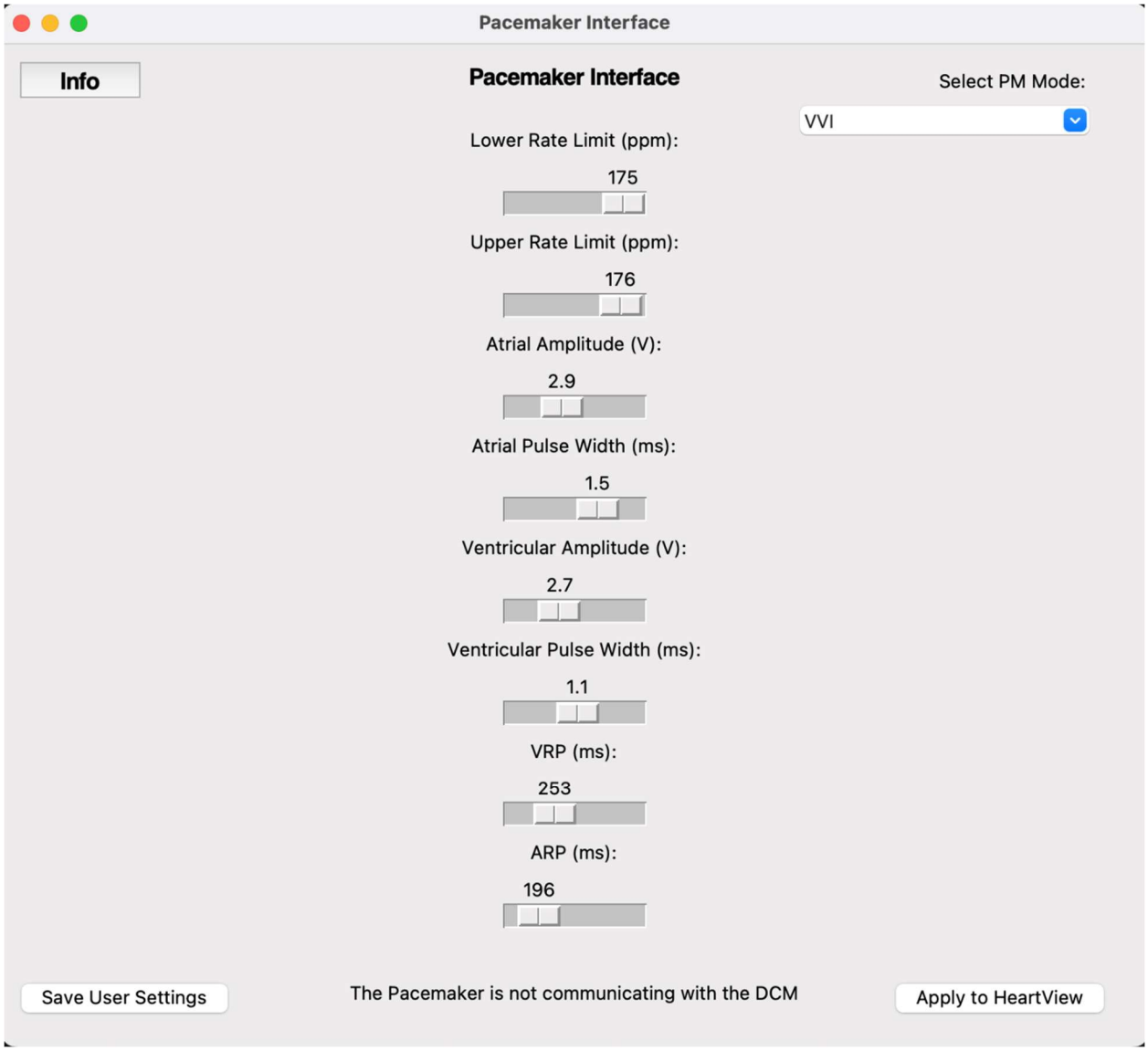


Figure 53: Main Interface

The above provides a clear view of the parameter interface, this being the main interface of the pacemaker UI. It includes slides for all the required parameters, only allowing the user to select an input within the required range. The sliders were programmed using a function to create each slider automatically promoting modularity within the code. The possibility of the user

accidentally setting the upper rate limit below the lower rate limit is evident therefore a function was added to prevent this, after this error was encountered through testing. The function can be found below, with comments present to briefly explain the function. It can also be seen that a message is displayed at the bottom of the application, telling the user whether or not the DCM is currently communicating with the pacemaker, as set in the DCM requirements.

```
def check_rate_limits(self, *args):
    lower_limit = self.sliders["Lower Rate Limit"].get() #retrieving the lower limit value
    upper_limit = self.sliders["Upper Rate Limit"].get() #retrieving the upper limit value

    if upper_limit < lower_limit:                #if upper less than lower, move upper to be one hgiher than lower
        self.sliders["Upper Rate Limit"].set(lower_limit + 1)
```

Figure 54: Rate limit verifcation function

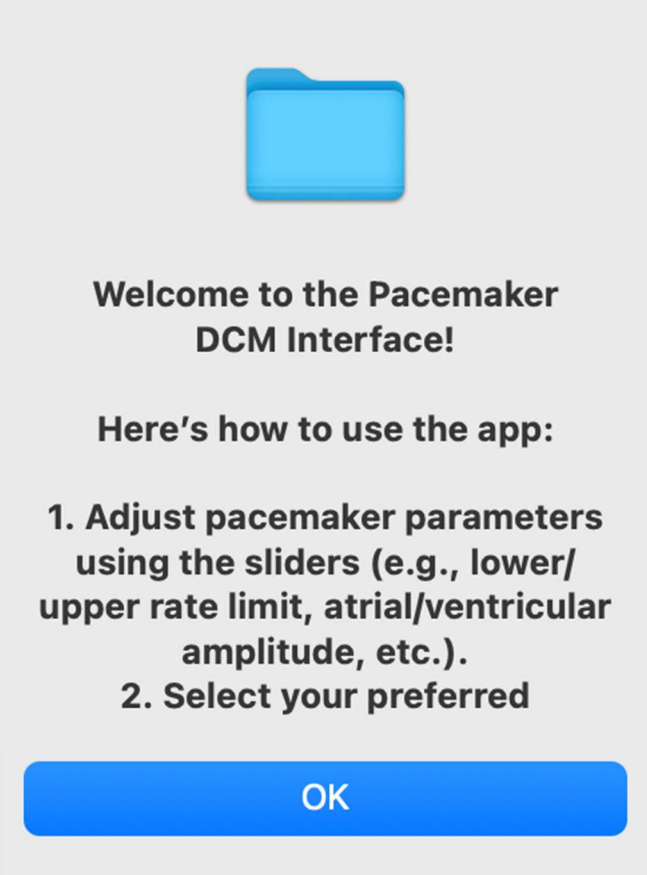


Figure 55: Information pop-up

The figure above presents the output of the info button found in the top right corner of the design. This feature was added after testing presented problems with new users who were unsure

of how the application functioned. It provides insight as to how the parameters can be manipulated and how the mode can be saved.

3.3.4 Heartview Connection Interface

The button shown in the bottom right of the screen was added in anticipation of assignment 2. It presents the user with a pop-up window displaying a message, indicating that once completed the Egram data will be present on this screen.

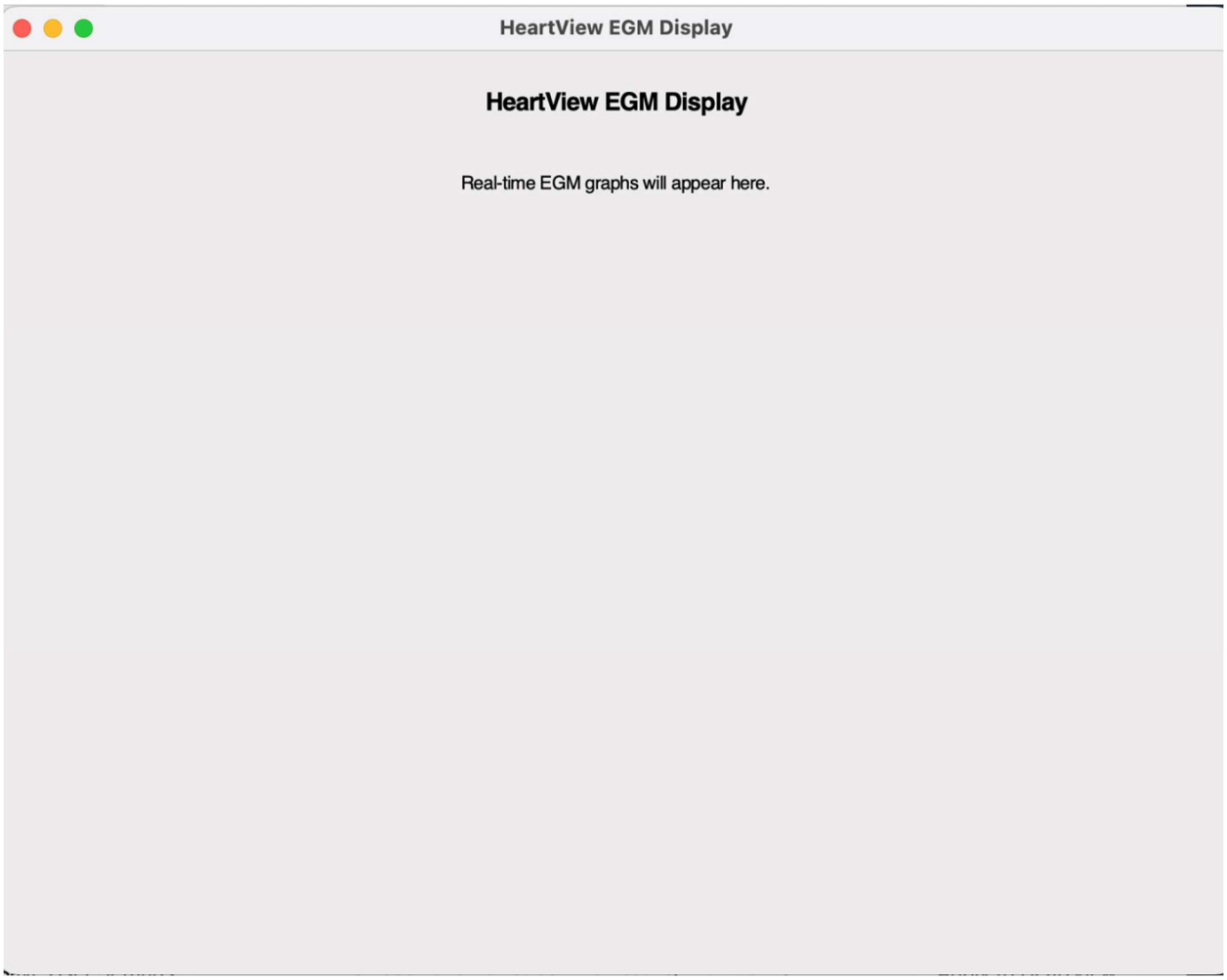


Figure 56: EGM window

```python
def open_pacemaker_interface(self, name):
    self.root.withdraw()   #removing the welcome screen
    pacemaker_window = tk.Toplevel(self.root)
    pacemaker_window.title("Pacemaker Interface")
    pacemaker_window.geometry("800x700")

    def on_close():
        self.root.destroy()
        pacemaker_window.destroy()  # Close the pacemaker window

    pacemaker_window.protocol("WM_DELETE_WINDOW", on_close) #bindin gthe exit button on the screen to closing the application


    title_font = tkFont.Font(family="Helvetica", size=16, weight="bold") #creating the title font and title
    tk.Label(pacemaker_window, text="Pacemaker Interface", font=title_font).pack(pady=10)

    slider_frame = tk.Frame(pacemaker_window)   #creating the frame where alll the sliders will go
    slider_frame.pack(pady=10, anchor="center")
```

Figure 57: Main interface code 1

```python
self.sliders = {}                          #creating sliders for all the paramters using the create_slider function
self.sliders["Lower Rate Limit"] = self.create_slider(slider_frame, "Lower Rate Limit (ppm):", 30, 175, callback=self.check_rate_limits)
self.sliders["Upper Rate Limit"] = self.create_slider(slider_frame, "Upper Rate Limit (ppm):", 60, 180, callback=self.check_rate_limits)

self.sliders["Atrial Amplitude"] = self.create_slider(slider_frame, "Atrial Amplitude (V):", 0.5, 7.0, resolution=0.1)
self.sliders["Atrial Pulse Width"] = self.create_slider(slider_frame, "Atrial Pulse Width (ms):", 0.1, 2.0, resolution=0.1)
self.sliders["Ventricular Amplitude"] = self.create_slider(slider_frame, "Ventricular Amplitude (V):", 0.5, 7.0, resolution=0.1)
self.sliders["Ventricular Pulse Width"] = self.create_slider(slider_frame, "Ventricular Pulse Width (ms):", 0.1, 2.0, resolution=0.1)
self.sliders["VRP"] = self.create_slider(slider_frame, "VRP (ms):", 150, 500)
self.sliders["ARP"] = self.create_slider(slider_frame, "ARP (ms):", 150, 500)

tk.Label(pacemaker_window, text="Select PM Mode:").place(relx=0.95, rely=0.02, anchor='ne')  #dropdown for slecting the mode
mode_options = ["AOO", "VOO", "AAI", "VVI"]
mode_dropdown = ttk.Combobox(pacemaker_window, values=mode_options, state="readonly")

if name in self.users:                                          #if name is in users we set their parameters as saved
    saved_parameters = self.users[name].get('parameters', {})
    for key in self.sliders:
```

Figure 58: Main interface code 2

```python
            if key in saved_parameters:
                self.sliders[key].set(saved_parameters[key])
        mode_dropdown.set(self.users[name]['mode'])
    else:
        mode_dropdown.set("AOO")

    mode_dropdown.place(relx=0.95, rely=0.06, anchor='ne')
                                                                #buttons for all the functions previouslt defined
    tk.Button(pacemaker_window, text="Info", command=self.show_info, font=title_font, width=6, height=1).place(x=10, y=10)
    tk.Button(pacemaker_window, text="Save User Settings", command=lambda: self.save_mode(name, mode_dropdown.get())).place(x=10, y=650)
    tk.Label(pacemaker_window, text="The Pacemaker is not communicating with the DCM").place(x=240, y=650)
    tk.Button(pacemaker_window, text="Apply to HeartView", command=self.show_heartview).place(x=620, y=650)

Initialize and run the application
icemakerApp()
```

Figure 59: Main interface code 3

The three figures above present the main logic for the interface. It shows the removal of the login window and how it is closed. It then moves into the formatting of the visuals the user will see including the sliders for all the parameters of the requirements. It also shows the creation

of the buttons shown on the interface and how they are positioned in specific positions to enhance the user experience. The rest of the functions are explained below in section 3.4.

3.3.5 OOP Implementation

When beginning Assignment 2, we decided to divide our code from a single file into many separate files to improve the readability, maintenance, and scalability of our DCM. This decision improved readability by organizing each class and its functionality into focused, standalone modules, making it easier to locate, understand, and work with specific components of the project. This decision also made maintenance easier by allowing us to adjust specific files or functions without affecting the rest of the code. It also allowed for us to add new pacemaker modes if needed without having to add or change other parts of the code, reducing the risk of introducing errors and simplifying updates. Lastly, dividing the code into multiple files improved scalability by enabling the seamless addition of new features or components without overloading a single file, keeping the project organized and manageable as it grows.

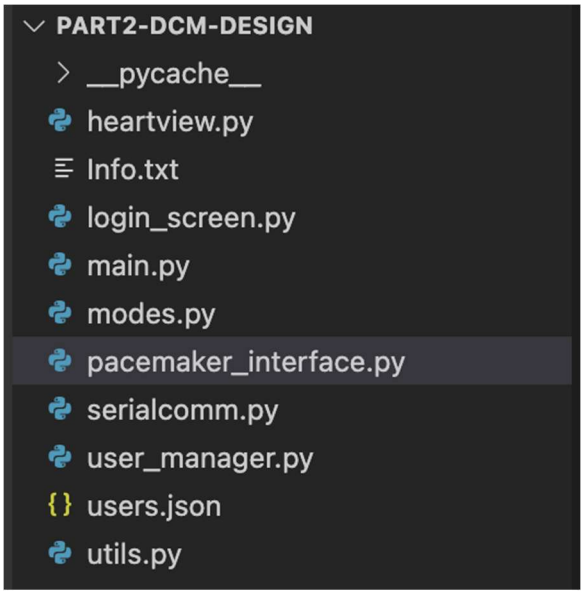


Figure 60: OOP Program files

3.3.6 eGram Display Window

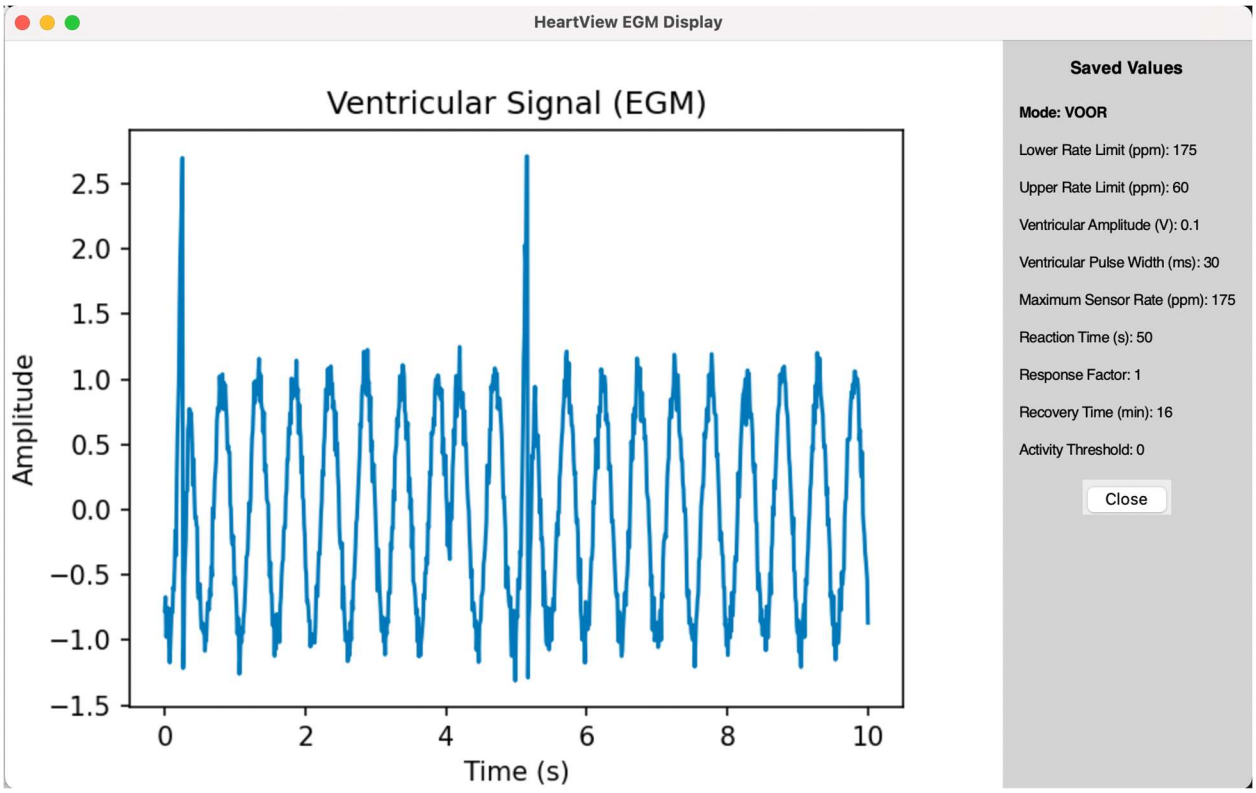


Figure 61: eGram Display

The above window provides a clear view of the eGram graph. It includes the list of set parameters and their values as well as the mode, so the user has all the necessary information at their disposal while viewing the graph. The DCM will send a packet of the applied pacemaker settings to the pacemaker, and it then receives a packet of data from the pacemaker which it uses to construct the graph displayed to the user.

3.3.7 Pacemaker Connection

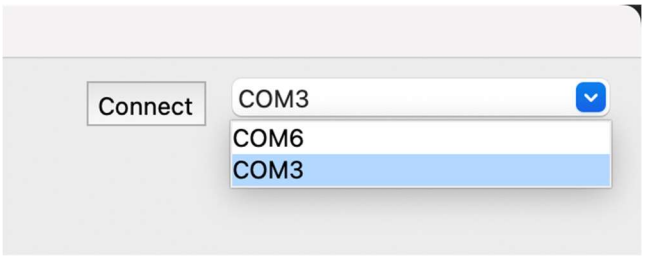


Figure 62:Connected Ports Dropdown

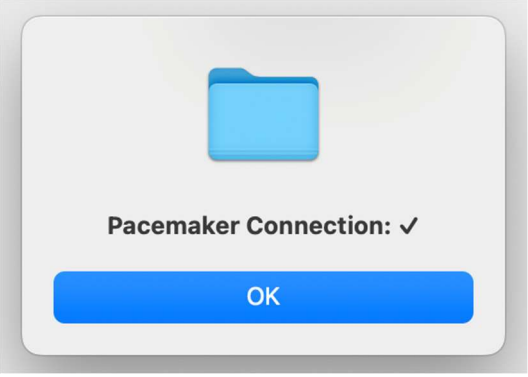


Figure 63: Pacemaker Connection Status

The above two figures demonstrate our pacemaker connection process and how it is displayed to the user. The user selects a connection port from the dropdown and when they click the connect button the message is displayed indicating the pacemaker connection status. This process helps the user verify that the pacemaker is connected properly before they begin inputting values and adjusting the parameters.

3.4 Module Overview

3.4.1 Purpose of the Modules

Assignment 1:

The Pacemaker App module is designed to provide a graphical user interface (GUI) for managing user logins and pacemaker settings. It allows users to register, log in, adjust pacemaker parameters, and view a simulated real-time Electrogram (EGM) display. The application uses secure password storage and limits the number of registered users.

Assignment 2:

For Assignment 2 we followed an OOP approach to creating our GUI. We broke our main class into multiple subclasses and folders that each contribute a part to the GUI.

HearView Class

The heartview.py file defines the HeartView class, which is responsible for creating a graphical interface to display a simulated Electrogram (EGM) signal, specifically a ventricular signal or atrial signal, along with user-saved cardiac parameters.

LoginScreen Class

The login_screen.py file defines the LoginScreen class, which provides a graphical user interface (GUI) for users to log in or register to access a pacemaker Direct Control Module (DCM) system.

Main.py

The main.py file serves as the entry point for the application. Its primary purpose is to initialize and display the LoginScreen, allowing users to authenticate and access the rest of the pacemaker Direct Control Module (DCM) system.

Modes.py

The modes.py file defines a dictionary, PARAM_FOR_MODES, which specifies the programmable parameters associated with different pacemaker operating modes. This file acts as a centralized configuration module for handling mode-specific settings.

PacemakerInterface Class

The PacemakerInterface class covers various features such as slider management, mode-specific configurations, serial communication and user settings.

SerialComm Class

The serial communication classes primary purpose is to send configuration parameters (derived from the JSON file) to the device via a UART (Universal Asynchronous Receiver/Transmitter) serial communication protocol, and to receive and parse response packets sent by the device.

UserManager Class

The usermanager.py script is a Python utility designed to manage user accounts for an application. Its purpose is to handle user registration, login verification, and the secure storage of user credentials and associated data.

Utility Class

The utils.py script provides utility functions that are mostly related to user management and password security. These functions are essential building blocks for a broader application.

3.4.2 List of Public Functions

3.4.2.1 Heart View Class

__init__(self): Initializes the application, sets up the window, and loads existing users.

load_users(self): Loads the user data from a JSON file.

save_users(self): Saves user data to a JSON file.

hash_password(self, password): Hashes the password using bcrypt.

check_password(self, hashed_password, password): Verifies if the provided password matches the hashed password.

register_user(self): Registers a new user if they don't exist.

login_user(self): Handles user login by verifying credentials.

clear_entries(self): Clears username and password entries.

show_info(self): Displays a message box with usage instructions.

check_rate_limits(self): Validates that the upper rate limit is greater than the lower rate limit.

create_slider(self): Creates a slider widget for adjusting values.

save_mode(self): Saves the selected mode and parameters.

show_heartview(self): Opens a new window to display heart EGM data.

open_pacemaker_interface(self): Opens a pacemaker settings interface.

3.4.2.2 Login Screen Class

__init__(self): Initializes the application and sets up the window.

create_widgets(self): Creates the login screen UI components.

register_user(self): Registers a new user with provided credentials.

login_user(self): Logs in a user by verifying credentials.

clear_entries(self): Clears the username and password fields.

open_pacemaker_interface(self, name): Opens the Pacemaker interface for the logged-in user.

on_close(self): Closes the application when the window is closed.

3.4.2.3 Serial Communication Class

read_user_parameters(filename, username): Reads parameters for a specific user from a JSON file.

create_packet(sync, fn_code, params): Creates a packet based on user parameters for serial communication.

receive_packet(ser): Receives and parses a packet from the serial port.

__main__(): Main execution block that initializes serial communication, sends a packet, and handles response parsing.

3.4.2.4 Pacemaker Interface Class

__init__(self, parent_window, username, user_manager): Initializes the pacemaker interface with the parent window, username, and user manager.

save_settings_sendData(self): Collects current settings, saves them, and sends the data to the pacemaker via UART.

show_heartview(self): Saves the current settings and opens the HeartView window for real-time data display.

sign_out(self): Logs the user out by closing the pacemaker interface and revealing the login screen.

on_close(self): Handles the closing of the pacemaker window and returns the user to the login screen.

create_sliders(self, frame): Creates sliders and labels for pacemaker parameters and places them in the given frame.

update_slider_visibility(self, event=None): Updates the visibility of sliders based on the selected mode, triggered by an optional event.

load_saved_settings(self): Loads and applies saved user settings to the sliders and dropdowns.

monitor_connection(self): Monitors the connection status by attempting to communicate with the pacemaker.

save_settings(self): Collects and saves the current settings to the user manager's storage.

show_info(self): Opens an information window detailing how to use the pacemaker interface.

3.4.2.5 User Manager Class

__init__(self): Initializes the application, sets up the window, and loads existing users.
load_users(self): Loads the user data from a JSON file.
save_users(self): Saves user data to a JSON file.
hash_password(self, password): Hashes the password using bcrypt.
check_password(self, hashed_password, password): Verifies if the provided password matches the hashed password.
register(self, name, password): Registers a new user if they don't exist.
login(self, name, password): Handles user login by verifying credentials.

3.4.2.6 Utils Class

load_users(): Loads the user data from a JSON file.

save_users(self, users): Saves user data to a JSON file.

hash_password(self, password): Hashes the password using bcrypt.

check_password(self, hashed_password, password): Verifies if the provided password matches the hashed password.

3.4.3 Black Box Behavior

3.4.3.1 HeartView Class

__init__

- **Input**: master, username, user_manager
- **Output**: Initializes a new HeartView window displaying the EGM signal and user parameters.
- **Behavior**:
    - Creates the GUI window with a graph for EGM data and a side panel displaying the user's saved values (e.g., rate limits and amplitude).
    - Fetches the user's parameters and mode.
    - Initializes and animates the ventricular signal plot using Matplotlib.

generate_ventricular_signal

- **Input**: None
- **Output**: A simulated ventricular signal (a noisy sinusoidal wave with R-peaks).
- **Behavior**:
    - Generates a sinusoidal ventricular signal influenced by the user's saved rate limits and amplitude.
    - Adds noise and simulated R-peaks to the signal for realistic EGM data.

update_plot

- **Input**: frame (frame number for animation)
- **Output**: Updates the graph with a shifted signal (scrolling effect).

- **Behavior**:
    - Rolls the ventricular signal by a fixed amount to create a scrolling effect, updating the plot with the new signal data.

close_heartview

- **Input**: None
- **Output**: Closes the HeartView window.
- **Behavior**: Destroys the window, effectively closing the HeartView interface.

3.4.3.2 Login Screen Class

__init__

- **Input**: root (Tkinter window)
- **Output**: Initializes the login screen UI with entry fields for username and password, and login/registration buttons.
- **Behavior**:
    - Creates a new Tkinter window (or uses the provided one).
    - Initializes the UserManager and sets up the window title and dimensions.
    - Calls create_widgets() to build the UI and starts the Tkinter event loop.

create_widgets

- **Input**: None
- **Output**: Sets up the UI elements (labels, entry fields, buttons).
- **Behavior**:
    - Creates labels for username and password fields.
    - Creates input fields for the user to enter their name and password.
    - Adds "Register" and "Login" buttons with respective actions.

register_user

- **Input**: None (retrieves input from entry fields)
- **Output**: Displays a message box showing the result of the registration attempt.
- **Behavior**:
    - Fetches the entered name and password.

    - Registers the user via UserManager and shows a message box with success or error.
    - Clears the input fields if registration is successful.

login_user

- **Input**: None (retrieves input from entry fields)
- **Output**: Displays a message box with the login result, either success or failure.
- **Behavior**:
    - Fetches the entered name and password.
    - Attempts to log the user in via UserManager.
    - If successful, shows a success message and opens the Pacemaker interface.
    - If unsuccessful, displays an error message.

clear_entries

- **Input**: None
- **Output**: Clears the content of the username and password input fields.
- **Behavior**:
    - Deletes the text in both the username and password fields.

open_pacemaker_interface

- **Input**: name (username)
- **Output**: Opens the Pacemaker interface and hides the login screen.
- **Behavior**:
    - Hides the login window by calling withdraw().
    - Opens the Pacemaker interface, passing the username and UserManager.

on_close

- **Input**: None
- **Output**: Closes the application.
- **Behavior**:
    - Stops the Tkinter event loop, effectively closing the login window and terminating the application.

3.4.3.3 Serial Communication Class

read_user_parameters(filename, username)

- **Input**:
  - filename: Path to the JSON file.
  - username: Username to look up.
- **Output**:
  - Dictionary of user parameters or None if the user is not found.
- **Behavior**:
  - Opens and reads the JSON file.
  - Searches for the specified username.
  - Returns the user's parameters if found, or prints an error if not found.

create_packet(sync, fn_code, params)

- **Input**:
  - sync: SYNC byte.
  - fn_code: Function code byte.
  - params: Dictionary of user parameters.
- **Output**:
  - A bytearray packet with user parameters.
- **Behavior**:
  - Initializes a packet.
  - Appends the SYNC byte, function code, and user parameters.
  - Packs floating-point and integer values into the packet.
  - Returns the complete packet.

receive_packet(ser)

- **Input**:
  - ser: Serial object.
- **Output**:
  - Prints received data or error message.
- **Behavior**:
  - Reads data from the serial port.
  - Validates the packet length.
  - Extracts values from the packet.
  - Prints parsed data or error message.

__main__()

- **Input**:
  - None (executed directly).
- **Output**:
  - Executes main logic, sends packet, and handles serial communication.
- **Behavior**:
  - Reads user parameters from a JSON file.
  - Initializes serial communication.
  - Sends packet and receives a response.
  - Closes the serial communication.

3.4.3.4 Pacemaker Interface Class

__init__(self, parent_window, username, user_manager)
Input:

- parent_window: Parent window object.
- username: Username of the logged-in user.
- user_manager: Manages user data and settings.
  Output:
- Initializes the pacemaker interface, prepares the UI, and assigns user data.
  Behavior:
- Sets up the user interface with the provided window and user data.
- Loads user-specific settings from user_manager.
- Prepares for interaction with the pacemaker and related settings.

save_settings_sendData(self)
Input:

- None (uses internal settings).
  Output:
- Sends collected settings to the pacemaker.
  Behavior:
- Gathers current settings from UI components (sliders, dropdowns).
- Packs these settings into a format suitable for transmission.
- Sends the data to the pacemaker via UART for updating.

show_heartview(self)
Input:

- None (uses internal settings).
  Output:
- Opens the HeartView window with real-time data from the pacemaker.
  Behavior:

- Saves current settings before proceeding.
- Launches the HeartView window for continuous pacemaker data visualization.

sign_out(self)
Input:

- None.
  Output:
- Logs the user out, closes the pacemaker interface, and redirects to the login screen.
  Behavior:
- Ends the current user session.
- Returns to the login interface, ensuring user data is cleared.

on_close(self)
Input:

- None.
  Output:
- Closes the pacemaker interface and redirects to the login screen.
  Behavior:
- Handles the closure of the application window.
- Ensures the user is redirected back to the login screen for reauthentication.

create_sliders(self, frame)
Input:

- frame: UI container to place the sliders in.
  Output:
- Creates sliders for pacemaker parameters and adds them to the given frame.
  Behavior:
- Dynamically generates sliders for each pacemaker setting.
- Places the sliders and their labels into the provided frame container for display.

update_slider_visibility(self, event=None)
Input:

- event: Optional event to trigger the update.
  Output:
- Updates which sliders are visible based on the selected mode.
  Behavior:
- Adjusts slider visibility dynamically based on selected pacemaker mode or other input criteria.
- Ensures only relevant sliders are visible to the user.

load_saved_settings(self)
Input:

- None.
  Output:
- Applies saved settings to the sliders and UI components.
  Behavior:
- Retrieves saved user settings from storage or the user manager.
- Updates UI elements (sliders, dropdowns) with the saved values.

monitor_connection(self)
Input:

- None.
  Output:
- Monitors pacemaker connection status and displays updates.
  Behavior:
- Continuously attempts to communicate with the pacemaker.
- Displays connection status (e.g., connected, disconnected, errors).

save_settings(self)
Input:

- None.

  Output:
- Saves current settings to persistent storage (user manager).

  Behavior:
- Collects current values from UI components.
- Saves these settings for future use, ensuring they persist across sessions.

show_info(self)

Input:

- None.

  Output:
- Opens an informational window displaying usage instructions for the interface.

  Behavior:
- Launches a separate window with details on how to interact with the pacemaker interface, providing user guidance.

3.4.3.5 User Manager Class

__init__(self)

Input: None

Output: Initializes the application with the root element and loads the existing users.

Behavior:

- Sets the root for the application.
- Calls load_users() to load the existing user data from a JSON file.
- If the file doesn't exist, an empty dictionary is returned.

load_users(self)

Input: None

Output: Returns a dictionary containing user data from the users.json file or an empty dictionary if the file is not found.

Behavior:

- Attempts to open and read the users.json file.
- If the file is found, the user data is loaded and returned as a dictionary.
- If the file is not found, an empty dictionary is returned.

save_users(self)
Input: None
Output: Saves the current user data to the users.json file.
Behavior:

- Opens the users.json file for writing.
- Writes the current self.users data to the file in a formatted JSON structure.

hash_password(self, password)
Input: password: The plain text password to be hashed.
Output: A hashed password string.
Behavior:

- Generates a salt using bcrypt.gensalt().
- Hashes the provided password using bcrypt and the generated salt.
- Returns the hashed password as a string.

check_password(self, hashed_password, password)
Input:

- hashed_password: The stored hashed password for comparison.
- password: The plain text password to verify.

Output: Returns True if the password matches the hashed password, otherwise False.
Behavior:

- Compares the plain text password with the stored hashed password using bcrypt's checkpw() method.
- Returns True if the passwords match, otherwise False.

register(self, name, password)
Input:

- name: The username to register.
- password: The plain text password for the user.

Output:

- True, "User registered successfully!" if registration is successful.
- False, "User already exists!" if the user already exists.
- False, "Maximum user limit reached!" if the user limit is exceeded.
- False, "Please enter both name and password." if either the name or password is missing.

Behavior:

- Checks if the username already exists in self.users.
- Verifies if the user count exceeds MAX_USERS.
- Hashes the password and adds the user to the self.users dictionary.
- Saves the updated user data to the file.

login(self, name, password)
Input:

- name: The username to login.
- password: The plain text password for authentication.

Output:

- True, "Login successful!" if the login is successful.
- False, "User does not exist." if the username is not found.
- False, "Incorrect password." if the password is incorrect.

Behavior:

- Checks if the username exists in self.users.

- If it exists, compares the provided password with the stored hashed password using check_password().
- Returns a success message if the login is successful, otherwise returns an error message.

3.4.3.6 Utils Class

load_users()

Input: None

Output: Dictionary of user data or an empty dictionary if the file doesn't exist.

Behavior: Opens the users.json file and loads the data. If the file is not found, returns an empty dictionary.

save_users(self, users)

Input: users: Dictionary containing user data to be saved.

Output: None

Behavior: Saves the provided users dictionary into the users.json file.

hash_password(self, password)

Input: password: A plaintext password string.

Output: A hashed password string.

Behavior: Generates a bcrypt salt and hashes the provided password, then returns the hashed password.

check_password(self, hashed_password, password)

Input: hashed_password: A bcrypt-hashed password.

Input: password: A plaintext password string.

Output: Boolean (True if the password matches, False otherwise).

Behavior: Compares the provided password with the hashed password and returns whether they match.

3.4.4 Global Variables

- **Users.JSON**: A dictionary that stores user data, with usernames as keys and their corresponding passwords and settings as values.
    - **Structure:**

```
json
{
  "username": {
    "password": "hashed_password",
    "mode": "AOO",
    "parameters": {
      "Lower Rate Limit": value,
      "Upper Rate Limit": value,
      ...
    }
  },
  ...
}
```

3.4.5 Private Functions

The current modules do not contain explicitly private functions, as all defined functions are public. However, functions like loading users, saving users, and hashing password could be considered private in the context of their usage within the modules.

3.4.6 Internal Behavior

3.4.6.1 Heart View Class

__init__

- Initializes the HeartView window and GUI layout.
- Fetches user data (username, user_manager) and initializes state variables like lower_rate_limit, upper_rate_limit, amplitude, and mode.
- Sets up the Matplotlib graph and animates the ventricular signal with FuncAnimation.

generate_ventricular_signal

- Generates a synthetic EGM signal based on the user's saved parameters.
- Creates a sinusoidal signal with noise and adds R-peaks (sharp spikes) to simulate a ventricular signal.

update_plot

- Shifts the EGM signal to the left using np.roll() to create a scrolling effect.
- Updates the plot with the new shifted signal.

close_heartview

- Closes the HeartView window by destroying it with self.window.destroy().

3.4.6.2 Login Screen Class

__init__

- Initializes the login screen UI and event loop.
- Creates a Tkinter window and sets up the window title and dimensions.
- Initializes UserManager to handle user registration and login functionality.
- Calls create_widgets() to generate the login form UI.
- Starts the Tkinter event loop with self.root.mainloop().

create_widgets

- Defines the fonts for the labels and buttons.
- Creates a label and entry field for the username input, storing the username entered by the user.
- Creates a label and entry field for the password input, hiding the password with the show="*" option.
- Creates "Register" and "Login" buttons, each bound to their respective callback methods (register_user and login_user).

register_user

- Retrieves the username and password entered by the user.
- Calls self.user_manager.register() to register the user with the provided credentials.
- Displays a message box with the result of the registration (success or failure).
- Clears the entry fields if registration is successful.

login_user

- Retrieves the username and password entered by the user.
- Calls self.user_manager.login() to check the credentials and log the user in.
- If login is successful, displays a success message and calls open_pacemaker_interface() to open the Pacemaker interface.
- If login fails, shows an error message.

clear_entries

- Clears the text in both the username and password input fields using delete(0, tk.END).

open_pacemaker_interface

- Hides the current login window using self.root.withdraw().
- Creates a new PacemakerInterface window and passes the username and UserManager instance to it, allowing the user to interact with the pacemaker settings.

on_close

- Ends the Tkinter event loop by calling self.root.quit(), effectively closing the login screen and terminating the application.

3.4.6.3 Serial Communication Class

read_user_parameters(filename, username)

- Opens the specified JSON file to read user data.
- Iterates through the file to find the matching username.
- If the username exists, it returns the user parameters as a dictionary.
- If not found, prints an error message and returns None.

create_packet(sync, fn_code, params)

- Initializes a packet as an empty bytearray.
- Adds the SYNC byte to the packet.
- Appends the function code to the packet.
- Iterates through the user parameters and appends their respective values in a formatted way:
    - Converts floating-point numbers to 4 bytes.
    - Converts integers to 2 bytes.
- Returns the constructed bytearray packet containing all data.

receive_packet(ser)

- Reads data from the serial port using the ser.read() method.
- Validates the packet's length to ensure it is at least 16 bytes long.
- If valid, unpacks the data from the packet and interprets it:
    - Extracts the SYNC byte, function code, and parameter data.
    - Converts the parameters to their correct types (e.g., integers, floats).
- If successful, prints the parsed data.

- If the packet is invalid, prints an error message.

__main__()

- Reads user parameters by calling read_user_parameters() with a specified JSON file and username.
- Initializes the serial connection and attempts to send data to the connected device.
- Calls create_packet() to generate the packet for transmission.
- Sends the generated packet via the serial port and waits for a response.
- After receiving the response, calls receive_packet() to process and display the received data.
- Finally, closes the serial connection when done.

3.4.6.4 Pacemaker Interface

__init__(self, parent_window, username, user_manager)

- Initializes the main interface for the pacemaker configuration tool.
- Sets the parent_window to create the window context, linking it to the larger application.
- Retrieves the username and user_manager to personalize settings and load user-specific data.
- Calls internal methods to prepare the UI, load user data, and set default values for pacemaker settings.

save_settings_sendData(self)

- Gathers all relevant user input (e.g., slider values, dropdown selections) from the UI.
- Checks for the presence of required parameters and validates them.
- Constructs a packet or data structure containing these settings.
- Serializes the settings and sends them over the communication interface (e.g., UART) to update the pacemaker's configuration.

show_heartview(self)

- Verifies if the settings have been saved before transitioning to the HeartView.
- Saves the current settings in case the user wants to return to them later.
- Initializes and opens the HeartView window, passing necessary data or parameters to display the pacemaker's real-time data and status.
- Optionally, starts a background process to handle data updates or live readings from the pacemaker.

sign_out(self)

- Clears any user-specific data, ensuring no sensitive information is left in the session.
- Terminates active connections with the pacemaker, if applicable, to prevent unauthorized access.
- Redirects the application flow back to the login screen, clearing the window and preparing for the next user login.

on_close(self)

- Handles the closing of the application window, ensuring all resources are properly released.
- Checks if the user has unsaved settings and prompts for confirmation before closing.
- Ensures the program exits gracefully, clearing memory and stopping any background processes.
- Redirects to the login screen or reinitializes the application state for the next session.

create_sliders(self, frame)

- Dynamically creates and configures slider UI components based on pacemaker settings.
- Each slider is bound to a specific parameter (e.g., heart rate, pacing mode).
- Configures the slider's range, step size, and initial value based on default or saved settings.
- Places the sliders within the provided frame container and ensures they are interactive and visually aligned.

update_slider_visibility(self, event=None)

- Based on the selected mode or user input, determines which sliders should be visible and which should be hidden.
- Dynamically updates the visibility of UI elements without reloading the interface, using event handlers or state variables.
- Redraws the interface to reflect changes in slider visibility.

load_saved_settings(self)

- Fetches the user's saved settings from a persistent storage mechanism (e.g., a local file, database, or user manager).
- Applies these settings to the appropriate UI components (e.g., sliders, dropdowns) to ensure the user sees their last configuration.
- If no saved settings are found, sets default values and notifies the user.

monitor_connection(self)

- Periodically checks the status of the pacemaker connection, using communication protocols like UART or Bluetooth.
- Displays connection status updates in real time (e.g., connected, disconnected, error messages).
- If the connection is lost, attempts to reconnect or alert the user with a relevant message.

save_settings(self)

- Retrieves all current settings from the UI (e.g., slider values, selected options).
- Validates the settings and formats them in a user-friendly structure for persistence.
- Saves these settings using the user_manager or a similar service to ensure they are stored for future use.
- Optionally, logs the save action or notifies the user upon successful saving.

show_info(self)

- Creates and displays a new window or dialog containing usage instructions for the pacemaker interface.
- The information may include details about the settings, pacemaker modes, and general application use.
- Allows the user to dismiss or interact with the information window, ensuring it does not block the main interface.

3.4.6.5 User Manager Class

__init__(self)

- Initializes the root and users attributes. Calls load_users() to load user data.

load_users(self)

- Attempts to open users.json and loads its contents into a dictionary. Returns an empty dictionary if the file is not found.

save_users(self)

- Serializes the users dictionary and writes it to users.json.

hash_password(self, password)

- Generates a bcrypt salt and hashes the password. Returns the hashed password.

check_password(self, hashed_password, password)

- Compares the input password with the stored hashed password using bcrypt's checkpw function.

register(self, name, password)

- Checks if the user already exists and if the user limit is reached. If not, hashes the password and stores the user with default settings.

login(self, name, password)

- Verifies if the user exists and checks the provided password against the stored hashed password. Returns success or error message accordingly.

3.4.6.6 Utils Class

load_users()

- Tries to open the users.json file in read mode.
- If the file exists, loads its content into a dictionary and returns it.
- If the file does not exist, catches the FileNotFoundError and returns an empty dictionary.

save_users(self, users)

- Opens the users.json file in write mode.
- Serializes the users dictionary into JSON format with indentation.
- Writes the serialized data into the file.

hash_password(self, password)

- Uses bcrypt's gensalt() method to generate a salt.
- Hashes the password using bcrypt.hashpw() and the generated salt.
- Returns the hashed password as a string after decoding from bytes.

check_password(self, hashed_password, password)

- Encodes the input password to bytes.
- Uses bcrypt.checkpw() to compare the provided password (encoded) with the stored hashed_password (also encoded).
- Returns True if they match, otherwise False.

3.5 Testing and Results

Testing was performed by manually verifying the user registration, login process, and parameter inputs. Each mode was selected, and the associated parameters were adjusted and stored locally. Future integration with the pacemaker hardware is planned for later stages.

Test Description	Input	Expected Result	Actual Result	Pass/Fail	Comments/Changes Needed
Register a new user with expected input	Name: "Alice", Password: "password123"	User registered successfully message displayed and moved on	User registered successfully message displayed and moved on	Pass	N/A
Attempt to register the same user twice	Name: "Alice", Password: "newpassword"	Error message "User already exists!" displayed.	Error message "User already exists!" displayed.	Pass	N/A
Register a user when user limit is reached	Name: "User10", Password: "pass123"	Error message "Maximum user limit reached!" displayed.	Error message "Maximum user limit reached!" displayed.	Pass	N/A
Attempt to register a	Name: "", Password: "password"	Error message "Please enter	Error message "Please	Pass	N/A

user without a name		both name and password."	enter both name and password."		
Attempt to login with correct credentials	Name: "Alice", Password: "password123"	Welcome message "Welcome Alice!" displayed.	Welcome message "Welcome Alice!" displayed.	Pass	N/A
Attempt to login with empty fields	Name: "", Password: ""	Error message "Please enter both name and password."	Error message "Please enter both name and password.	Pass	N/A
Check rate limits functionality	Lower Limit: 60 Upper Limit: 50	Upper limit should automatically set to 61.	Upper limit set to 60.	Fail	Update the check_rate_limits logic to enforce limits.
Attempt to save user mode with	Mode: ""	Error message "Mode	Mode saved	Fail	Ensure mode selection cannot be left blank

empty mode value		cannot be empty."			
Set lower limit at 0 bpm	Lower Limit: 0, Upper Limit: 100	Error message "Lower limit cannot be zero."	Limits set	Fail	Establish reasonable minimums for limits
Test sign out feature	Press “Sign Out” button on pacemaker UI	Returns user back to login page	Returned to login page	Pass	N/A
Check if ports are displayed in dropdown when pacemaker is connected	Connect pacemaker and open port selection dropdown	COM3 and COM6 ports should appear in dropdown	COM3 and COM6 ports appear in the dropdown	Pass	N/A
Test connect button while connected to pacemaker	Press connect button on pacemaker UI	Display message: “Pacemaker Connection: ✓”	Displayed message: “Pacemaker Connection: ✓”	Pass	N/A

Test connect button while not connected to pacemaker	Press connect button on pacemaker UI	Display message: “Pacemaker Connection: X”	Displayed message: “Pacemaker Connection: X”	Pass	N/A
Check if only required sliders appears when selecting a mode	Select pacemaker mode from box dropdown (for all pacemaker modes)	Only display sliders that apply to that specific pacemaker mode	Only displays the sliders that apply to specific pacemaker mode (all modes)	Pass	N/A
Attempt to apply to heartview without saving	Press “Apply to Heartview” button without saving settings	Save settings and display message before opening heartview	Settings are saved and message is displayed before heartview opens	Pass	N/A
Open Egram graph	Press button to open Egram graph	Separate window appears with Egram graph	Window appears but no graph is displayed	Fail	Verify receive_packet() function is working properly

Open Egram graph	Press button to open Egram	Separate window with Egram graph appears	Egram graph appears with proper output	Pass	N/A
Open Egram with changed pacemaker settings	Save pacemaker settings and press button to open Egram	Egram window appears and reflects adjusted parameters	Egram appeared and reflects the adjusted parameters	Pass	N/A

Table 21: DCM Testing

3.6 Potential Changes and Future Considerations

3.6.1 Requirements Changes

Assignment 1:

In the future, there are many changes that will be required. Due to the increase in Simulink modes, parameters, and other factors, there will be a rise in the number of requirements for the DCM in the future. Since the DCM will be required to communicate directly with the pacemaker for assignment 2, many requirements which were not present previously will come into play.

The user interface will be required to be capable of visually indicating when telemetry is lost due to the device being out of range. This is crucial because if this was ever presented in a real-life situation, quick action would be required to ensure the connection is not lost and the pacemaker can still perform its job. Along with this, the UI must also be capable of visually indicating when telemetry is lost due to noise. For the same reasons as before this is equally as crucial.

The DCM will need to include an About function to provide essential system details, displaying the application model and version, DCM serial number, and the institution's name. It will also need to contain a Set Clock function that enables users to configure the device's date and time for accurate logging. Along with this, a New Patient function, where a new device can be connected and integrated without requiring the application to restart, streamlining patient management. Lastly, a Quit function will be required to allow for a telemetry session to be ended at any time, without causing any issues and freeing up system resources for other patients.

3.6.2 Design Changes

Assignment 1:

As a DCM is constantly changing, there will always be design changes that could be made to improve user experience and the application's ability to complete its job of providing safety and security to its users.

Currently, user data is stored in a JSON file, which can limit the scalability and security of the system. A future design enhancement could involve integrating a larger database solution, such as SQL, to handle larger data volumes and support more complex data streams. This upgrade would improve data management and improve performance, particularly if the DCM has a growing user base. A database solution could also allow multiple users to have access to the platform at once, which would be essential for expanding the application's capabilities.

When incorporating EGM data into the system, creating a dedicated page for its visualization would provide users with real-time insights into the patient's cardiac rhythms. The page currently present for the improvements in this area could feature customizable graphs, alerts, or markers for critical heartbeat events, allowing users to analyze EGM data directly within the application. These features would not only enhance the DCM's usability but also support better decision-making for doctors and patients.

To continuously improve the user experience, collecting feedback through surveys would be beneficial. By gathering insights on which areas users find challenging, these areas can be prioritized to directly address weak points. This feedback-driven approach allows for improvement in areas that may be looked over by the team to be targeted, improving accessibility, and overall usability. Implementing these changes would align the application's evolution with user needs, making it more intuitive and efficient for a wider range of users.

In future iterations of the DCM, implementing more OOP (object-oriented programming) would be crucial for improving code organization and maintainability. By dividing newly developed code into classes and reusable components, the application can become increasingly modular, making future code easier to implement. An OOP approach also helps with testing as it can be easier to create isolated, modular tests for each component. OOP would also reduce redundancy and coupling, some factors that are important to limit for the code to be understandable for other professionals, aside from just programmers.

Assignment 2:

To enhance the security of the DCM, future iterations will incorporate password requirements aligned with best practices observed across modern applications. These requirements may include a minimum length (e.g., 8 characters), the inclusion of at least one uppercase letter, one lowercase letter, one numeral, and one special character. Such measures will limit unauthorized access and improve the overall security posture of the system. Additionally, password expiration protocols and restrictions against reusing old passwords can be implemented to maintain long-term security. Visual aids such as progress bars to indicate password strength will be included to improve user experience.

As pacemaker technology evolves, the DCM should accommodate newer operational modes to ensure continued support for advanced devices. Future updates will include the ability to manage and configure additional modes beyond the currently supported parameters, such as multi-chamber pacing or adaptive pacing modes. This will involve expanding the parameter

input interface, testing safety constraints for the new modes, and ensuring seamless integration into the existing system. By adapting to emerging pacemaker technologies, the DCM can remain a cutting-edge tool for clinicians and patients alike.

To make the DCM more inclusive, support for multiple languages will be introduced. The interface will provide translations for English, Danish, Dutch, French, German, Spanish, Italian, and Swedish. A language selection menu will be implemented during setup, and all user interactions, including menus, instructions, and error messages, will be dynamically translated. Additionally, language-specific fonts and formatting will be considered for optimal display. This feature will allow healthcare providers and patients worldwide to interact with the DCM comfortably, reducing language barriers and improving usability.

To improve record-keeping and analysis, the DCM will feature enhanced reporting capabilities. Users will be able to generate and print various reports, including Bradycardia Parameters, Temporary Parameters, Implant Data, Threshold Test Results, Measured Data, Marker Legend, Session Net Change, and a Final Report combining key insights. Reports will be available in multiple languages to meet the needs of global users. Moreover, customization options will allow users to specify the type and format of reports (e.g., PDF, printed copies, or digital summaries). These reports will provide valuable insights for clinicians, aiding in monitoring, diagnosis, and decision-making.

These enhancements aim to make the DCM more secure, adaptable, inclusive, and functional for a broader range of users and applications.

3.7 DCM Development History

3.7.1 Assignment 1: Development History Chart

Date	Changes
September 29, 2024	Initialized the DCM application using the Tkinter library.

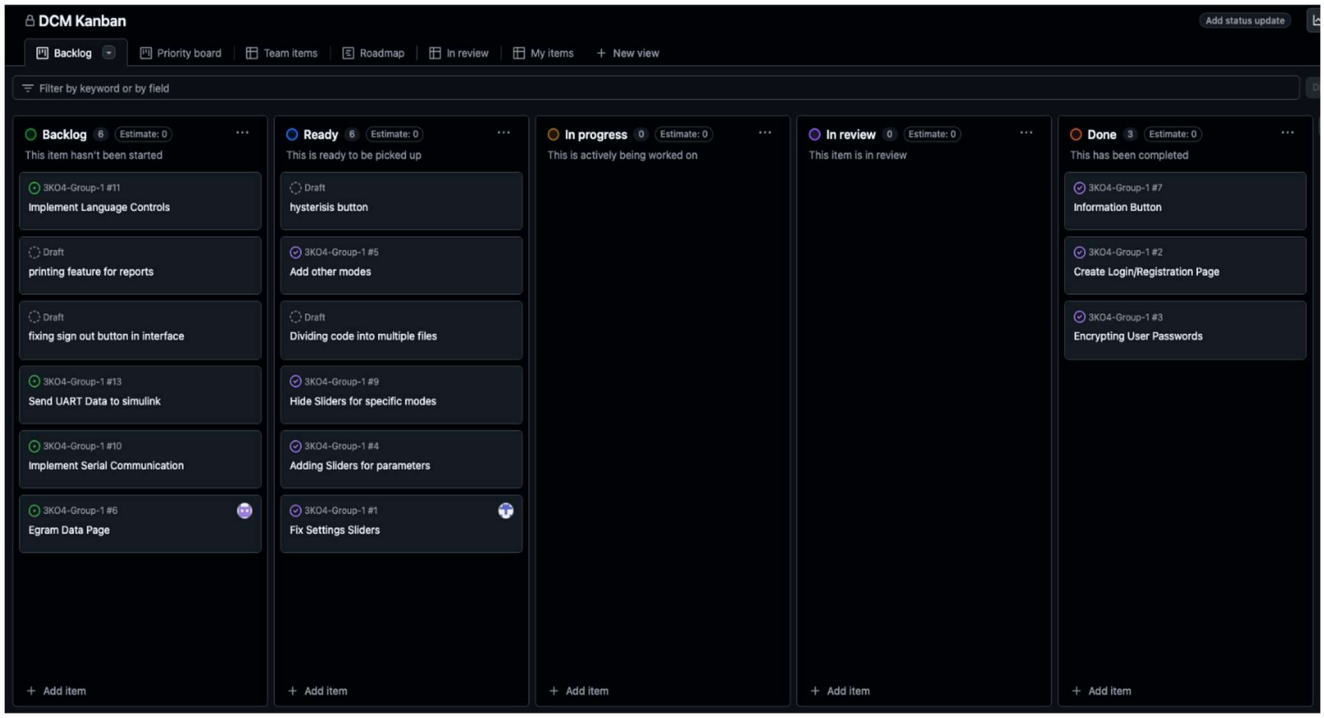
	Defined constants for maximum users (MAX_USERS = 10) and the user data file (USERS_FILE = 'users.json'). Created the PacemakerApp class, handling user login and registration.
October 4, 2024	Created the login interface with "Name" and "Password" fields. Configured "Register" and "Login" buttons to call register_user and login_user methods, respectively.
October 7, 2024	Implemented load_users and save_users functions to manage user data in JSON format. Verified correct functionality for user registration, including unique username checks and the maximum user limit.
October 10, 2024	Set up the main pacemaker interface to display adjustable parameters, including rate limits, atrial and ventricular amplitudes, and pulse widths. Added sliders with real-time validation (check_rate_limits) to ensure logical parameter relationships. Integrated mode selection with a dropdown menu, defaulting to "AOO." Enabled dynamic saving of the selected mode and slider parameters per user profile using the save_mode function.
October 16, 2024	Created the show_info function to guide users in adjusting pacemaker parameters. Added a "HeartView" visualization placeholder for displaying EGM data as required, confirming functionality for parameter retrieval, display, and saved settings reloading on subsequent logins.

October 22, 2024	Validated the complete functionality of the DCM application, ensuring user access and parameter saving/recall features work seamlessly. Added bcrypt for secure password hashing; implemented functions hash_password and check_password.

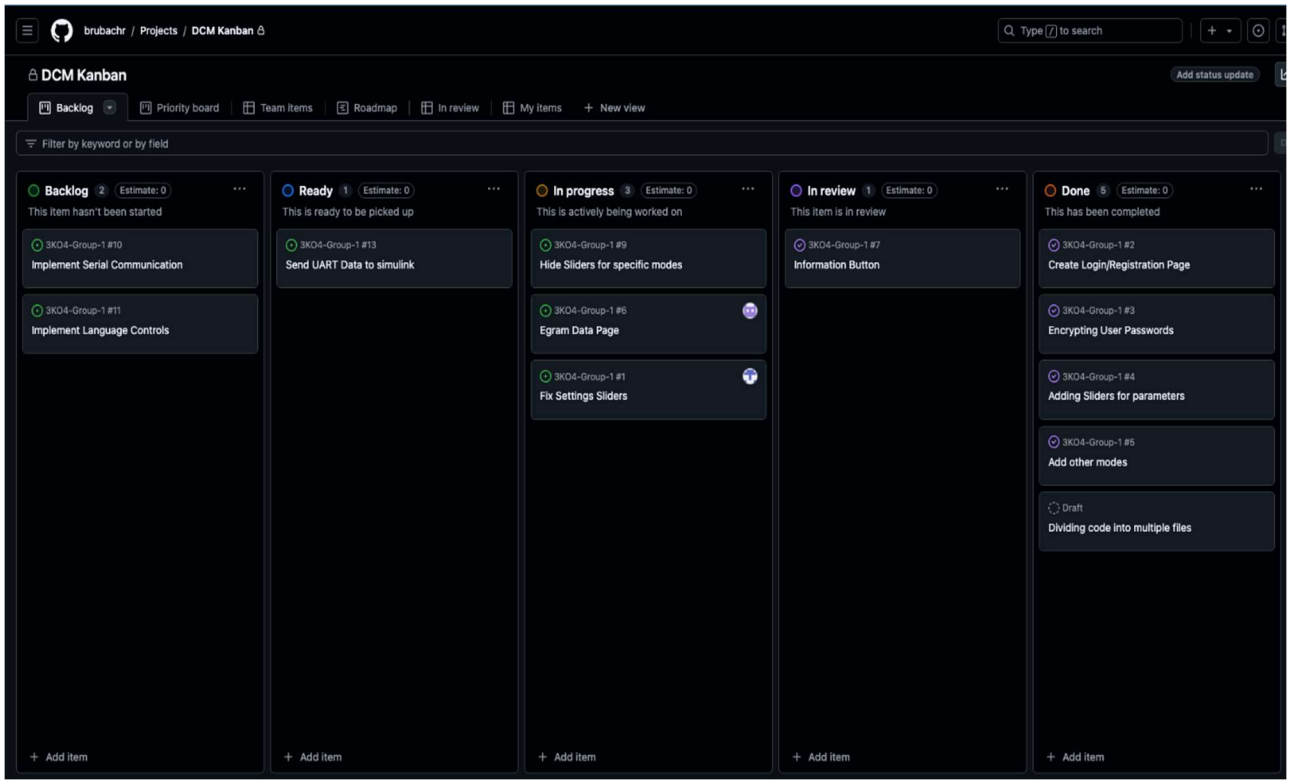
Table 22: DCM Design Log

3.7.2 Assignment 2: Kanban Board

Following Assignment 1, our group recognized that our original chart for tracking and recording design changes was disorganized and difficult to interpret. As a result, we decided to adopt a Kanban board for managing and documenting all design decisions and changes throughout the remainder of the pacemaker project. This approach leverages our previously developed Agile project management skills, enabling us to create a more structured and effective development workflow.

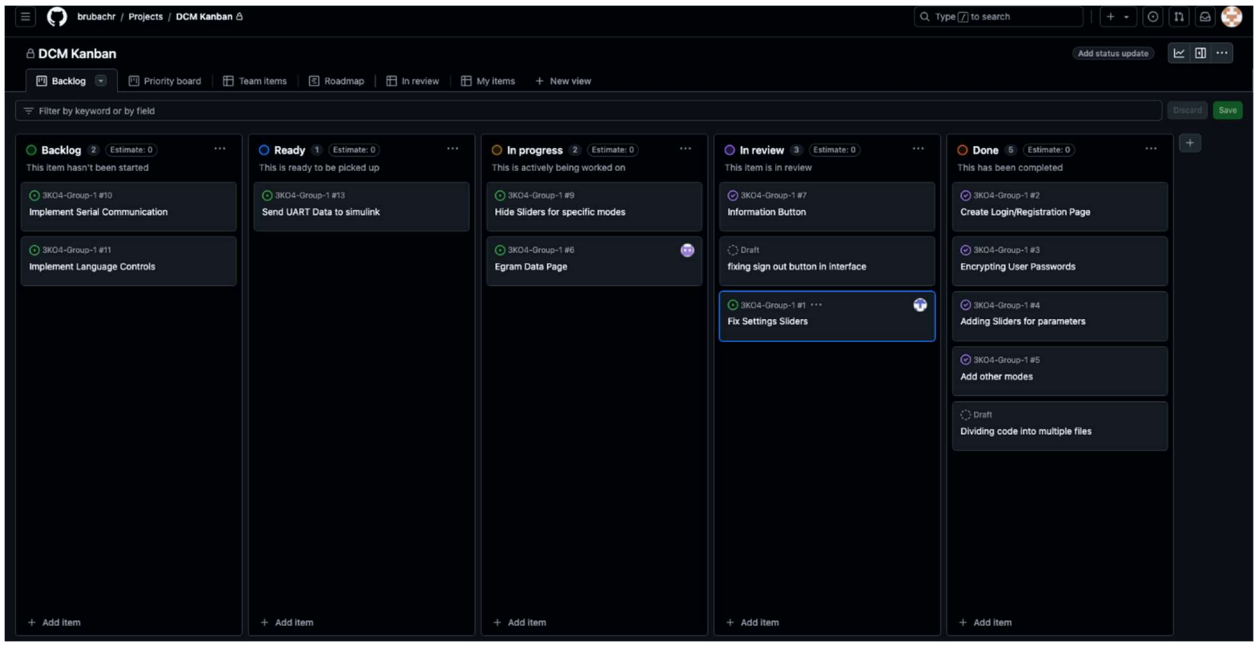


The above displays the Kanban board at the beginning of assignment 2, on Nov 1st, 2024. The Backlog contains items we will develop in assignment 2 but have not yet investigated. The Ready column contains the tasks we have planned and are ready for development. The Done column shows the items we have already completed in Assignment 1. Changes will be made to the Kanban to show the progress of the pacemaker development.

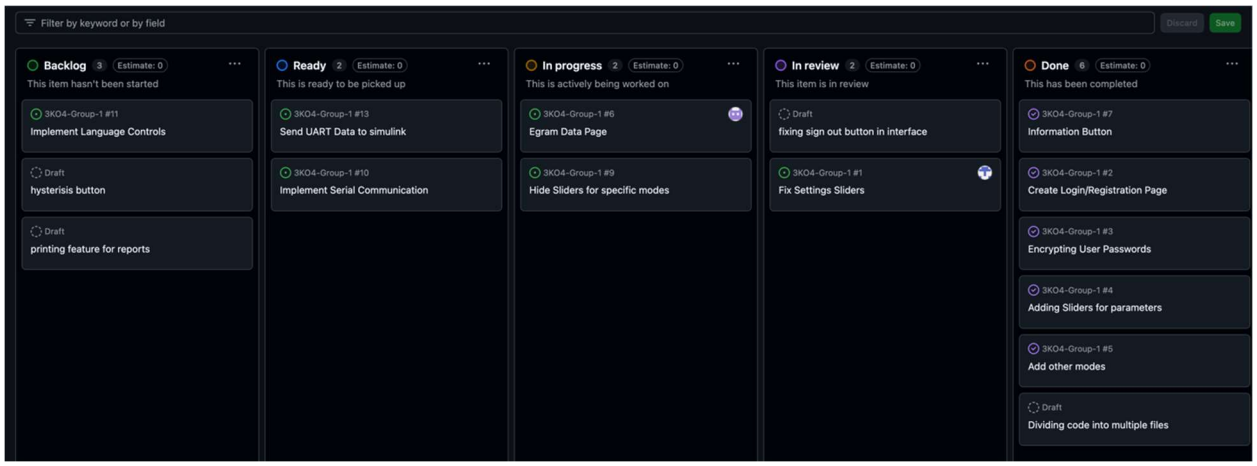


The Kanban board has now been updated after a meeting and work session was conducted on November 11th, 2024. During this session, we worked on reviewing the information button, which was previously believed to be completed but later had some additions. During this session, sliders were added for the new parameters introduced in assignment 2, and the four new modes were also added to the mode selection drop-down and the modes.py file.

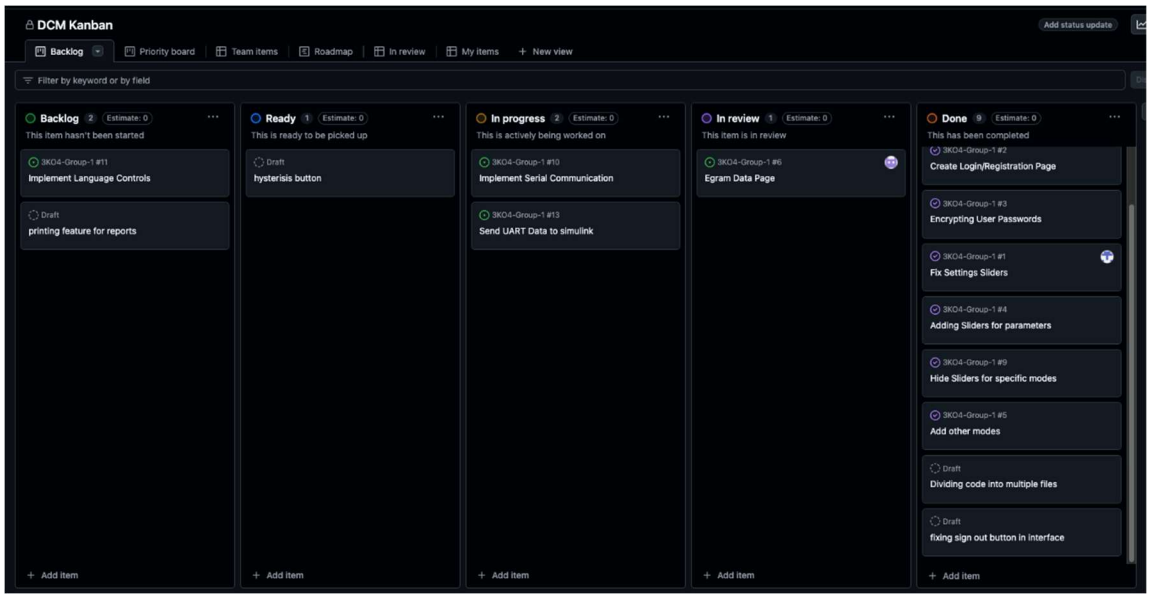
A large task was completed during this session, dividing the preexisting single main python file into 5 tailored classes for specific parts of the DCM interface, including the main.py, heartview.py, pacemaker_interface.py, user_manager.py, and modes.py.



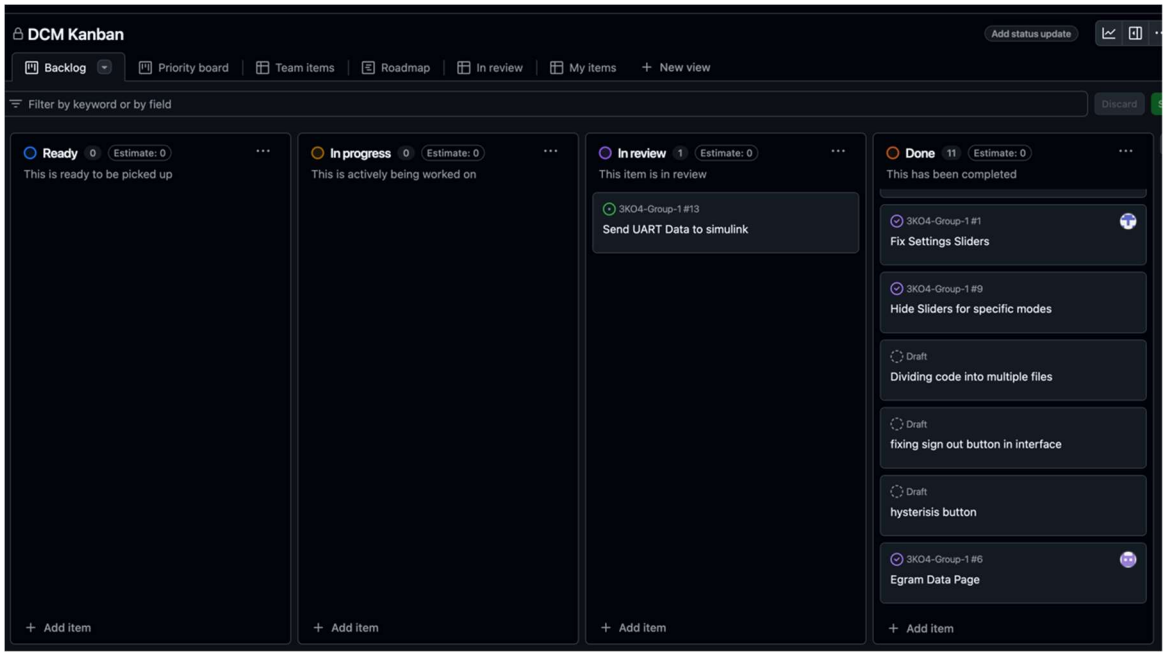
The next work meeting was conducted on Nov 15th, 2024. During this work session, one team member continued working on hiding sliders for specific modes, which was a feature that was forgotten during assignment 1, which was required. We also worked on fixing the settings sliders to prevent user input errors for the ranges. No tasks were finalized during this period however lots of valuable work was conducted.



On Nov. 22nd, 2024, a work session was conducted to add some new tasks which included the hysteresis button, to allow the user to decide if this parameter would be used, and the information button was reviewed and completed.



On Nov. 25th, 2024, a meeting was conducted to finish lots of the remaining tasks. This included fixing the sign-out button, hiding and fixing the sliders for specific modes, and the hysteresis button began development.



Our last session for the pacemaker DCM team was held on Nov. 27th, 2024, the day before the demo. During this time, we finalized the remaining tasks, including the hysteresis button, and the Egram data page. The sending of UART data and serial communication were combined into one task which was reviewed during the meeting and finalized that night. It was decided that the language and printing tasks would not be completed to prioritize the more important features of the design.

4. Assurance Case

4.1 – Top Claim

Claim: The pacemaker, in combination with the Device Communication Module (DCM), provides reliable, safe, and effective cardiac rhythm management.

Assumptions:

- The pacemaker is used to treat patients with specific cardiac conditions, such as bradycardia, and is prescribed by a licensed healthcare provider in accordance with medical guidelines.
- The pacemaker is implanted and programmed by a qualified, trained medical professional, and its performance is monitored regularly during follow-up visits according to established clinical protocols.
- The pacemaker is manufactured according to applicable regulatory standards (e.g., ISO, FDA), and it is properly inserted into the patient's body following established surgical procedures.
- The patient follows the prescribed treatment plan, including adhering to post-implantation care and attending necessary follow-up appointments to ensure the pacemaker functions properly.

Context:

- The pacemaker is designed to regulate heart rhythms, particularly for patients with bradycardia or other cardiac arrhythmias.
- The DCM facilitates post-implantation adjustments of the pacemaker's pacing settings.

Argument:

- All potential hazards and defects associated with the pacemaker have been thoroughly identified and effectively mitigated.
- All potential hazards and defects related to the Device Communication Module (DCM) have been systematically identified and addressed.
- All communication-related hazards and defects between the pacemaker and the DCM have been meticulously mitigated, ensuring seamless and secure data exchange between devices.

4.1.1 Top Subclaim

Subclaim: The Simulink model of the pacemaker has been rigorously validated to ensure its safety and security for the intended use across the following operational modes: VOO, AOO, VVI, AAI, VOOR, AOOR, VVIR, and AAIR.

Assumptions:

- The device is used exclusively for its intended purposes under all operational conditions.
- All sensors, components, and connections of the device remain fully functional and undamaged throughout the device's lifespan.
- The Simulink model remains unchanged and continues to operate as it was originally validated, without modification or degradation over time.

Context:

- The Simulink model controls the main process of the pacemaker’s modes, sensing and UART communication with the DCM.

Arguments:

- All potential hazards and risks associated with the Simulink model have been thoroughly identified and effectively mitigated through rigorous safety analysis and validation processes.
- The requirements outlined in section 2.2 of this document are fully implemented and adhered to for each operational mode to ensure compliance with industry standards and safety protocols.

4.1.1.1 Subclaim

Subclaim: The Simulink model ensures that the lower rate limit and maximum sensor rate are never exceeded or fallen below while in a rate adaptive pacing mode.

Assumptions:

- The pacemaker's intended operating conditions, including heart rate thresholds, are clearly defined and adhered to.
- The sensors are calibrated to accurately monitor and respond to physiological signals within the prescribed rate limits.

Context:

- The pacemaker’s pacing mode is dynamically adjusted based on the sensed heart rate, and the Simulink model enforces these thresholds to prevent the device from entering unsafe operational conditions.

Arguments:

- The Simulink model has been validated through rigorous testing to ensure that the LRL and MSR are never violated, even in edge-case scenarios where heart rate or sensor readings fluctuate.
- Safety mechanisms have been integrated within the model to guarantee that the pacemaker will automatically adjust pacing when the heart rate nears either limit, thereby maintaining safe operation.

Evidence:

- Test results show that, across all validated operational modes, the pacemaker’s pacing output remains within the defined LRL and MSR limits.
- Test cases demonstrate that the pacemaker adjusts its pacing mode when the heart rate approaches or exceeds the preset rate thresholds, maintaining safe operation and patient health.

4.1.1.2 Subclaim

Subclaim: The Simulink system uses separate state flow charts for each mode, ensuring that one mode will not affect the other.

Assumptions:

- Each mode requires their own state logic, tailored to its intended functionality.
- Transitions between modes occur under controlled conditions based on the DCM and heart rate.

- The state flow charts have clear boundaries and do not share resources that may lead to interference and error.

Context:

- The pacemaker allows for multiple pacing modes based on the users' cardiac conditions, ensuring safe and reliable function. To achieve this, the Simulink system utilizes modular state flow charts, with each mode operating independently. This design prevents unwanted interaction between modes, ensuring appropriate actions and avoiding malfunctions.

Arguments:

- By isolating each mode, faults in one mode will not interfere or compromise the functionality of other modes.
- Separate state flows make the overall system easier to test, debug, and validate, as each mode does not directly interact with another.

Evidence:

- The failed test cases show that when one mode has an issue, the other modes will be unaffected.
- When running AOO with a faulty state flow parameter, running VOO after still works. This proves that the modularity protects one mode from malfunctioning if another mode has errors.

4.1.1.3 Subclaim

Subclaim: The Simulink model correctly implements all requirements listed in section 2.2 for each mode of operation.

Assumptions:

- The requirements outlined in section 2.2 are comprehensive and represent the full set of functional and safety specifications for the pacemaker across all modes of operation.
- The pacemaker's sensors, components, and connections are fully functional during testing and simulation.
- The board generating the heartbeat is creating realistic physiological conditions during testing.

Context:

- The pacemaker operates in multiple modes (e.g., VOO, AOO, VVI, AAI, VOOR, AOOR, VVIR, and AAIR), each with specific functional requirements such as rate limits, pacing width, amplitude, and activity detection.

Argument:

- The Simulink model has been rigorously tested to ensure that all requirements defined in section 2.2 are met in all operational modes. This includes verifying that the pacemaker responds correctly to sensor inputs, adheres to pacing limits, and adjusts pacing rates as required in rate adaptive modes.
- All sensing functionality has been tested and confirmed through multiple rounds of testing and simulation analysis.

Evidence:

- Testing performed on all 8 modes of operation (VOO, AOO, VVI, AAI, VOOR, AOOR, VVIR, and AAIR) shows successful operation even with multiple different input parameters (Lower rate limit, upper rate limit, pace width and amplitude).
- Pacemaker successfully senses and inhibits pacing in AAI and VVI modes when a heartbeat is detected, as confirmed through testing results.
- Acceleration is successfully detected and mapped to activity level, resulting in a corresponding increase in pacing rate in rate adaptive modes, confirmed through testing.

4.1.2 Top Subclaim

Subclaim: The design and implementation of the Device Communication Module (DCM) has been rigorously validated to ensure its safety, security, and reliability for all user data while maintaining seamless accessibility.

Assumptions:

- The DCM is used exclusively for its intended purposes under all operational conditions.
- All components, software modules, and the user interface of the DCM remains fully functional and undamaged throughout its lifespan.

Context:

- The Device Control Monitor (DCM) serves as the front-end user interface, enabling users to configure parameters and select pacemaker modes.
- The DCM ensures user focus by displaying only the essential information required for operation, hiding unnecessary details.

Strategy:

- Test all possible states and inputs of the DCM through extensive regression testing

Arguments:

- The login functionality ensures secure user access through password encryption, hidden input, and structured data storage.
- Parameter sliders enforce safety by restricting input values to within predefined and validated ranges.
- The eGram page enhances user awareness by providing clear and real-time visualization of parameter settings and pacemaker mode.
- The DCM fully adheres to the requirements outlined in Section 3.2.2, ensuring it meets all industry standards for functionality, safety, and user feedback.

4.1.2.1 Subclaim

Subclaim: The login functionality ensures user data is securely managed through password hiding, encryption, organized data storage, and limited user accounts.

Assumptions:

- Passwords are entered and stored securely, preventing unauthorized access.
- The maximum number of allowed user accounts is enforced to avoid system overload.

Context:

- Login credentials are stored and organized using a JSON file structure, with user passwords encrypted for added security.

Arguments:

- Password hiding during input ensures privacy while logging in.
- Encryption of passwords prevents unauthorized access to user credentials.
- Limiting user accounts maintains system performance and prevents unauthorized users from accessing the system.
- Organized data storage in JSON allows efficient management of user profiles.

Evidence:

- Testing confirms that unauthorized login attempts are blocked, and encrypted passwords cannot be deciphered without access to the main program.
- Testing confirms that when a maximum of 10 users have been entered no more users will be allowed to sign up for the system.

4.1.2.2 Subclaim

Subclaim: The provided parameter sliders ensure that parameter input values remain within safe and predefined ranges.

Assumptions:

- Parameter ranges are predetermined based on safety and operational requirements.
- Sliders are calibrated to prevent inputs outside of these ranges.
- Sliders are calibrated to ensure no conflicts between upper and lower bounded sliders.

Context:

- The DCM interface uses sliders to allow users to adjust parameters intuitively while maintaining safe operational limits.

Arguments:

- Input validation prevents users from exceeding or falling below safe values.
- The sliders are user-friendly and reduce the likelihood of manual input errors.

Evidence:

- Validation tests confirm that sliders prevent values from going outside the safe operational range. This was tested by testing all sliders to ensure their values can not exceeed their required ranges.
- Testing ensured that the upper rate limit can not surpass the lower rate limit.

4.1.2.3 Subclaim

Subclaim: The eGram page displays user parameters and pacemaker modes to ensure users are aware of their programmed parameters.

Assumptions:

- Displayed parameters accurately reflect user inputs.
- The eGram page is updated in real-time when the user exits and modifies their settings.

Context:

- The eGram page provides a graphical and textual representation of active modes and user-defined settings.

Arguments:

- Displaying parameter values increases user awareness and reduces the risk of misconfiguration.
- Real-time updates provide immediate feedback, allowing users to validate their inputted parameter changes.

Evidence:

- Usability testing confirms that user parameters are accurately displayed and, when changed, are accurately adjusted accordingly. This was tested by manipulating all different parameters across the eight possible modes resulting in all changes to the parameters being displayed correctly.

- Testing through different user accounts ensured that parameter changes are recorded to the given user's particular account and their account only.

4.1.2.4 Subclaim

Subclaim: The DCM meets all requirements specified in Section 3.2.2 for user interface functionality.

Assumptions:

- The DCM operates under intended conditions and interfaces correctly with other system components.
- All specified features in Section 3.2.2 are correctly implemented and tested.

Context:

- The DCM interface provides the necessary features to manage windows, process inputs, display parameters, and visually indicate system states.

Arguments:

- The DCM can manage windows for displaying text and graphics.
- The interface processes user inputs, including positioning and button interactions.
- All programmable parameters can be displayed for review and modification.
- The DCM provides visual indications for communication status, including when the DCM is connected to the pacemaker.
- The interface visually alerts users when the pacemaker device is approached.

Evidence:

- Functional tests demonstrate that the DCM meets all Section 3.2.2 requirements under various operational scenarios. The DCM windows may all be managed accordingly, and tested through verification of opening and closing all possible windows to ensure the program functions correctly.

4.1.3 Top Subclaim

Subclaim: The UART communication between the Simulink model and the DCM is safe and secure, while effectively transmitting information.

Assumptions:

- The DCM is only used by a trained professional and the correct parameters are selected for transmission to the pacemaker.
- The UART line between the pacemaker and DCM is secure, with no risk of interception or tampering during transmission.
- The DCM and pacemaker hardware function as designed, with no physical or electronic faults affecting communication.

Context:

- UART (Universal Asynchronous Receiver-Transmitter) communication is used to transmit critical data between the pacemaker and DCM, such as programming parameters, operational settings, and device status updates.

Argument:

- The Simulink model has been rigorously tested to verify compliance with UART communication standards, ensuring that all transmitted data is accurate and error-free.
- Security features have been implemented on both ends of the communication channel to ensure the integrity and correctness of transmitted data
- The Simulink model incorporates fallback settings to handle scenarios where data is corrupted, incorrect, or synchronization with the DCM is not achievable. These mechanisms ensure that pacemaker operation continues safely under such conditions.

4.1.3.1 Subclaim

Subclaim: The Simulink model only changes its modes when specifically set by the DCM.

Assumptions:

- The DCM sends valid and verified mode change commands.
- Communication between the DCM and the Simulink model is reliable, with no interruptions or data corruption.

- The Simulink model maintains its current mode unless a specific mode change command is received and processed correctly.

Context:

- The system operates in various pacing modes crucial for simulating pacemaker behavior.
- Dynamic mode changes are required for testing and validation under controlled conditions.
- Restricting mode changes to DCM commands enhances safety, reliability, and traceability.

Argument:

- Limiting mode changes to explicit DCM commands ensures predictable and safe operation.
- This approach simplifies debugging and validation by maintaining traceability for all mode transitions.
- Preventing internal overrides or unintended transitions ensures the system aligns with safety and functional requirements.

Evidence:

- The Stateflow diagram explicitly links parameter changes and mode transitions to specific function codes (rxdata(2)) received from the DCM.
- Logic within the model prevents transitions from the initial safe mode to bradycardia therapy without the DCM sending the required command.
- Simulations show that the initial mode reliably avoids therapy activation until the DCM explicitly allows it, validating the safety-focused design.

4.1.3.2 Subclaim

Subclaim: The Simulink model confirms the data that is sent by the DCM by echoing back parameters for verification.

Assumptions:

- The DCM verifies that the echoed parameters match the originally transmitted values.
- The Simulink model echoes back parameters immediately after they are received and applied, ensuring synchronization.

- Communication between the DCM and Simulink model is reliable and supports two-way data exchange without corruption or delays.

Context:

- The DCM is the primary controller responsible for configuring the pacemaker model with programmable parameters.
- Echoing the parameters ensures that the pacemaker model correctly interprets and applies the intended configurations sent by the DCM.
- This verification mechanism enhances system safety by detecting communication errors or misconfigurations early.

Argument:

- The Simulink model is designed to echo back every parameter set received from the DCM after successfully applying it.
- This feedback loop allows the DCM to cross-check and validate that the transmitted parameters match the echoed ones, ensuring correctness.
- By verifying the echoed parameters, the system reduces the risk of unintended behaviors caused by data corruption or misinterpretation.

Evidence:

- The Stateflow diagram includes a dedicated ECHO_PARAM state that calls the send_params() function to echo all received parameters back to the DCM for validation.
- Simulation tests demonstrate that the model correctly echoes back all parameters immediately after applying them, without modifications or omissions.
- Annotations in the Simulink model clearly outline the steps involved in echoing and verifying the parameters, ensuring traceability and alignment with the assurance claim.

5. Conclusion

This document provides a comprehensive overview of the design, development, and testing of the pacemaker modes and the DCM interface. The system successfully meets all requirements outlined in Assignments 1 and 2, ensuring safety, functionality, and compliance with specified standards. Furthermore, the design is adaptable, leaving room for the incorporation of future requirements and enhancements.