

Pacemaker Project

Assignment 2

MECHTRON 3K04: Software Development

December 3rd, 2023

Harry Easton (400390139), Gianluca Capuano (400374041), Jonathan Hughes (40036583), Bhavyakumar Shah (400394087), Jainil Majmudar (400373281), Liviru Abeygunawardena (400389917)

Table of Contents

Pacemaker and Simulink Design Documentation.....	9
Pacemaker and Simulink Requirements	9
Pacemaker and Simulink Safety Assurance Case Diagram:	10
Design Decisions and Model Information	11
Pacemaker Modes	11
Sensing Circuit Design.....	12
Timing Chart Design.....	13
Pacemaker Pacing Design	16
Rate Adaptive Calculation and Accelerometer Activity	16
Serial Communication.....	19
Pacemaker and Simulink Testing.....	21
AOO Testing	22
VOO Testing	26
AAI Testing	30
VVI Testing	39
AOOR Testing	48
VOOR Testing	49
AAIR Testing.....	50
VVIR Testing.....	53
Future Requirement Changes.....	56
Future Design Changes	56
DCM Design Documentation	57
DCM Requirements	57
DCM Safety Assurance Case Diagram.....	57
DCM Design Decisions and Information.....	59
Login & Registration	59
Pacemaker Selection	62
Mode Selection	65
Displaying Existing Data	69
Placeholder for Electrogram Data.....	70
Further Improvements.....	71

Changes Made To user_manager.py	73
Changes Made To pacemaker_interface.py	75
Addition of New Modes & Changes Made To mode_sel.py	75
Serial Communication.....	77
Future Requirement Changes.....	81
Future Design Changes	81
Explanation of Modules with Testing.....	82
mode_sel.py:	82
pacemaker_interface.py:	89
main.py:	97
user_manager.py:	101
display.py:	103
egram.py:	108
serial_communication.py:	114
Appendix A: Simulink Sensing Circuit Model	117
Appendix B: Simulink Timing Circuit Model	118
Appendix C: Simulink Pacing Circuit Model	119
Appendix D: Simulink Rate Adaptive and Accelerometer Subsystem.....	120
Appendix E: Serial Communication	122

Table of Figures:

Figure 1: Accelerometer Subsystem in Simulink	16
Figure 2: desired behavior for decreasing activity.....	18
Figure 3: desired behavior for increasing activity.....	18
Figure 4: the Heartview output for Test 1; testing the AOO mode at 60bpm and a target voltage of 3V. .	22
Figure 5: the Heartview output for Test 2; testing the AOO mode at 120bpm and a target voltage of 1V.	23
Figure 6: the Heartview output for Test 3; testing the AOO mode at 60bpm and a target voltage of 10V.	24
Figure 7: the Heartview output for Test 4; testing the AOO mode when button SW2 is pressed.	25
Figure 8: the Heartview output for Test 1; testing the VOO mode at 60bpm and a target voltage of 3V. .	26
Figure 9: the Heartview output for Test 2; testing the VOO mode at 30bpm and a target voltage of 5V. .	27
Figure 10: the Heartview output for Test 3; testing the VOO mode at 60bpm and a target voltage of 15V.	28
Figure 11: the Heartview output for Test 4; testing the VOO mode when button SW2 is pressed.	29
Figure 12: the Heartview output for Test 1; testing the AA1 mode at 60bpm while heart is at 30bpm. ...	30
Figure 13: the Heartview output for Test 2; testing the AA1 mode at 60bpm while heart is at 60bpm. ...	31
Figure 14: the Heartview output for Test 3; testing the AA1 mode at 60bpm while heart is at 59bpm. ...	32
Figure 15: the Heartview output for Test 4; testing the AA1 mode at 60bpm while heart is at 61bpm.	33
Figure 16: the Heartview output for Test 5; testing the AA1 mode at 60bpm while heart is at 120bpm. .	34

Figure 17: the Heartview output for Test 6; testing the AA1 mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.....	35
Figure 18: the Heartview output for Test 7; testing the AA1 mode at 60bpm while heart is at 30bpm and a target voltage of 10V.....	36
Figure 19: the Heartview output for Test 8; testing the AA1 mode at 60bpm with an ARP of 150ms while heart is at 30bpm.....	37
Figure 20: the Heartview output for Test 9; testing the AA1 mode at 60bpm with an ARP of 500ms while heart is at 30bpm.....	38
Figure 21: the Heartview output for Test 1; testing the VV1 mode at 60bpm while heart is at 30bpm. ...	39
Figure 22: the Heartview output for Test 2; testing the VV1 mode at 60bpm while heart is at 60bpm. ...	40
Figure 23: the Heartview output for Test 3; testing the VV1 mode at 60bpm while heart is at 59bpm. ...	41
Figure 24: the Heartview output for Test 4; testing the VV1 mode at 60bpm while heart is at 61bpm. ...	42
Figure 25: the Heartview output for Test 5; testing the VV1 mode at 60bpm while heart is at 120bpm. .	43
Figure 26: the Heartview output for Test 6; testing the VV1 mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.....	44
Figure 27: the Heartview output for Test 7; testing the VV1 mode at 60bpm while heart is at 30bpm and a target voltage of 10V.....	45
Figure 28: the Heartview output for Test 8; testing the VV1 mode at 60bpm with an VRP of 150ms while heart is at 30bpm.....	46
Figure 29: the Heartview output for Test 9; testing the AA1 mode at 60bpm with an ARP of 500ms while heart is at 30bpm.....	47
Figure 30: Testing AOOR with a LRL of 60bpm and URL of 120bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	48
Figure 31: Testing VOOR with a LRL of 60bpm and URL of 120bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	49
Figure 32: Testing AAIR with a LRL of 60 bpm and URL of 120bpm while the heart is at 90bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	50
Figure 33: Testing AAIR with a LRL of 60bpm and URL of 120bpm while the heart is at 30bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	51
Figure 34: Testing AAIR with a LRL of 60bpm and URL of 120bpm while the heart is at 140bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	52
Figure 35: Testing VVIR with a LRL of 60bpm and URL of 120bpm while the heart is at 60bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	53
Figure 36: Testing VVIR with a LRL of 60bpm and URL of 120bpm while the heart is at 30bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	54
Figure 37: Testing VVIR with a LRL of 60bpm and URL of 120bpm while the heart is at 140bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).	55
Figure 38: FIGMA Model of the sign in interface.....	60
Figure 39: Implementation of Login Page Based on FIGMA Model	60

Figure 40: Implementation of Registration Page Based on FIGMA Model	61
Figure 41: Fix of Using Register Entry instead of Login Entry.....	61
Figure 42: Fix for Routing Issue for Functions with Multiple Arguments	62
Figure 43: Circular Import Error.....	62
Figure 44: Sending an instance of the main module into the constructor of submodule	62
Figure 45: Initializing submodules with the main module as an argument	62
Figure 46: GUI for Pacemaker Selection Screen	63
Figure 47: Warning Popup for Different Pacemaker Connection.....	63
Figure 48: The solution for whitespace submissions	64
Figure 49: Error Message for Blank Pacemaker Input.....	64
Figure 50: Instance of mode_sel.py	65
Figure 51: Demonstration of AOO Parameter Rendering.....	65
Figure 52: Slider Getting Stuck on Small Value of 0.2	66
Figure 53: Error Label Implementation	66
Figure 54: Iterative Solution of Error Checking	67
Figure 55: Message Box Implementation for Invalid Parameter Input.....	68
Figure 56: Final GUI for Mode Selection with all Elements	69
Figure 57: GUI for the Display Existing Data Frame	69
Figure 58: Electrogram Data Placeholder Graph.....	70
Figure 59: reset_mode_sel function.....	71
Figure 60: New Display Screen	72
Figure 61: User Manager Class.....	73
Figure 62: New Modes Shown in the Dropdown Menu	76
Figure 63: Opening JSON with Error Handling	77
Figure 64: All Data Storage Files Used	78
Figure 65: Sending Data Packet.....	79
Figure 66: Unpacking Received Data	80
Figure 67: Verification of Received Data	80
Figure 68: Error Label not Showing for Upper Rate Limit	84
Figure 69: Error Label Malfunctioning Due to Modulus Logic	85
Figure 70: Test 1 of modulus with 0.1	85
Figure 71: Test 2 of modulus with 0.5	86
Figure 72: All error labels display	87
Figure 73: Pop Up error warning functionality.....	87
Figure 74: Error Label functionality	88
Figure 75: Previous Pacemaker Showing None (Not Saving)	90
Figure 76: Previous Pacemaker Saving & Updating Label	91
Figure 77: Not Throwing Error/Allowing Connection for Blank Pacemaker Input	91
Figure 78: Fixed Error Message that blocks blank pacemaker input	92
Figure 79: Commented out line that had entry clearing upon leaving pacemaker selection page	92
Figure 80: Warning about difference pacemaker connection	93
Figure 81: Successful Connection Message.....	93
Figure 82: Successful Submission of Same Pacemaker.....	94
Figure 83: Pacemaker Interface when Pacemaker Connected	94
Figure 84: Pacemaker Interface when Pacemaker Disconnected.....	95
Figure 85: Warning showing the user cannot proceed further without connecting Pacemaker.....	95
Figure 86: New Heart Board Warning.....	96

Figure 87: New Pacemaker Board Warning	96
Figure 88: Error of Maximum Users Reached.....	98
Figure 89: Username already in data storage error	99
Figure 90: Error Message of Different Password Submissions During Registration	99
Figure 91: Error Message when Non Existing User Info is Entered.....	100
Figure 92: Error thrown when registering with a clear user_data file	100
Figure 93: JSON Decoding Error This was due to the string variable being used in the dropdown returning a 'NoneType' which the JSON decoder didn't accept. To fix this, we added a conditional shown below:.....	104
Figure 94: Conditional that checks if no pacemakers exist in data.....	105
Figure 95: Error message shown when no pacemaker exists.....	105
Figure 96: Command to make label wrap when 1000pixels of width covered.....	105
Figure 97: Data Submission.....	106
Figure 98: Data displayed	106
Figure 99: Visible data but showing excess towards the right.....	107
Figure 100: Showing the ability to scroll showing more submitted data.....	107
Figure 101: Placeholder Graph for Electrogram Data	110
Figure 102: Correct Egram Pulses on Graph for VOO Mode.....	111
Figure 103: Check Pacemaker Connection Error Message.....	111
Figure 104: Graph Mode Dropdown List	112
Figure 105: Two Graphs for Atrial + Ventricular	113
Figure 106: Disconnect Pacemaker before Pressing Submit	115
Figure 107: Saves Pacemaker Data for New User	115
Figure 108: Successful Communication with Pacemaker.....	116
Figure 109: Unsuccessful Communication with Pacemaker	116
Figure 110: Sensing Circuit Exterior	117
Figure 111: Sensing Circuit Interior	117
Figure 112: Timing Chart Outside	118
Figure 113: Timing Chart Initial State	118
Figure 114: Pacing Circuit Outside.....	119
Figure 115: Pacing Circuit Inside	119
Figure 116: Rate Adaptive Exterior	120
Figure 117: Rate Adaptive Calculation.....	120
Figure 118: Accelerometer Subsystem Exterior	121
Figure 119: Accelerometer Interior.....	121
Figure 120: Scaling and Averaging Calculations.....	121
Figure 121: Serial Communication Outside.....	122
Figure 122: Serial Communication Inside	122
Figure 123: Serial Communication Output Outside.....	123
Figure 124: Serial Out and byte packaging.....	124

Table 1: the parameters and corresponding integers for the modes included in the Simulink model (AOO, VOO, AAI, and VVI).....	11
Table 2: Input and output corresponding pin names.....	12
Table 3: Timing chart variables with corresponding pacemaker variables.....	13
Table 4: Variable layout for AOO and AAI charging state.	14
Table 5: Variable layout for AOO and AAI pacing state.....	14
Table 6: Rate adaptive inputs.....	17
Table 7: Input parameter datatypes.....	19
Table 8: Input parameter nominal values.....	20
Table : the pacemaker and Heartview inputs for Test 1; testing the AOO mode at 60bpm and a target voltage of 3V.....	22
Table : the pacemaker and Heartview inputs for Test 2; testing the AOO mode at 120bpm and a target voltage of 1V.....	23
Table : the pacemaker and Heartview inputs for Test 3; testing the AOO mode at 60bpm and a target voltage of 10V.....	24
Table : the pacemaker and Heartview inputs for Test 4; testing the AOO mode at 60bpm and a target voltage of 5V when the button is switched off.	25
Table : the pacemaker and Heartview inputs for Test 1; testing the VOO mode at 60bpm and a target voltage of 3V.....	26
Table : the pacemaker and Heartview inputs for Test 2; testing the AOO mode at 30bpm and a target voltage of 5V.....	27
Table : the pacemaker and Heartview inputs for Test 3; testing the VOO mode at 60bpm and a target voltage of 10V.....	28
Table : the pacemaker and Heartview inputs for Test 4; testing the VOO mode at 60bpm and a target voltage of 5V when the button is pushed.	29
Table : the pacemaker and Heartview inputs for Test 1; testing the AAI mode at 60bpm while heart is at 30bpm.	30
Table : the pacemaker and Heartview inputs for Test 2; testing the AAI mode at 60bpm while heart is at 60bpm.	31
Table : the pacemaker and Heartview inputs for Test 3; testing the AAI mode at 60bpm while heart is at 59bpm.	32
Table : the pacemaker and Heartview inputs for Test 4; testing the AAI mode at 60bpm while heart is at 61bpm.	33
Table : the pacemaker and Heartview inputs for Test 5; testing the AAI mode at 60bpm while heart is at 120bpm.	34
Table : the pacemaker and Heartview inputs for Test 6; testing the AAI mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.	35
Table : the pacemaker and Heartview inputs for Test 7; testing the AAI mode at 60bpm while heart is at 30bpm and a target voltage of 10V.....	36
Table : the pacemaker and Heartview inputs for Test 8; testing the AAI mode at 60bpm with an ARP of 150ms while heart is at 30bpm.	37
Table : the pacemaker and Heartview inputs for Test 9; testing the AAI mode at 60bpm with an ARP of 500ms while heart is at 30bpm.	38
Table : the pacemaker and Heartview inputs for Test 1; testing the VVI mode at 60bpm while heart is at 30bpm.	39
Table : the pacemaker and Heartview inputs for Test 2; testing the VVI mode at 60bpm while heart is at 60bpm.	40

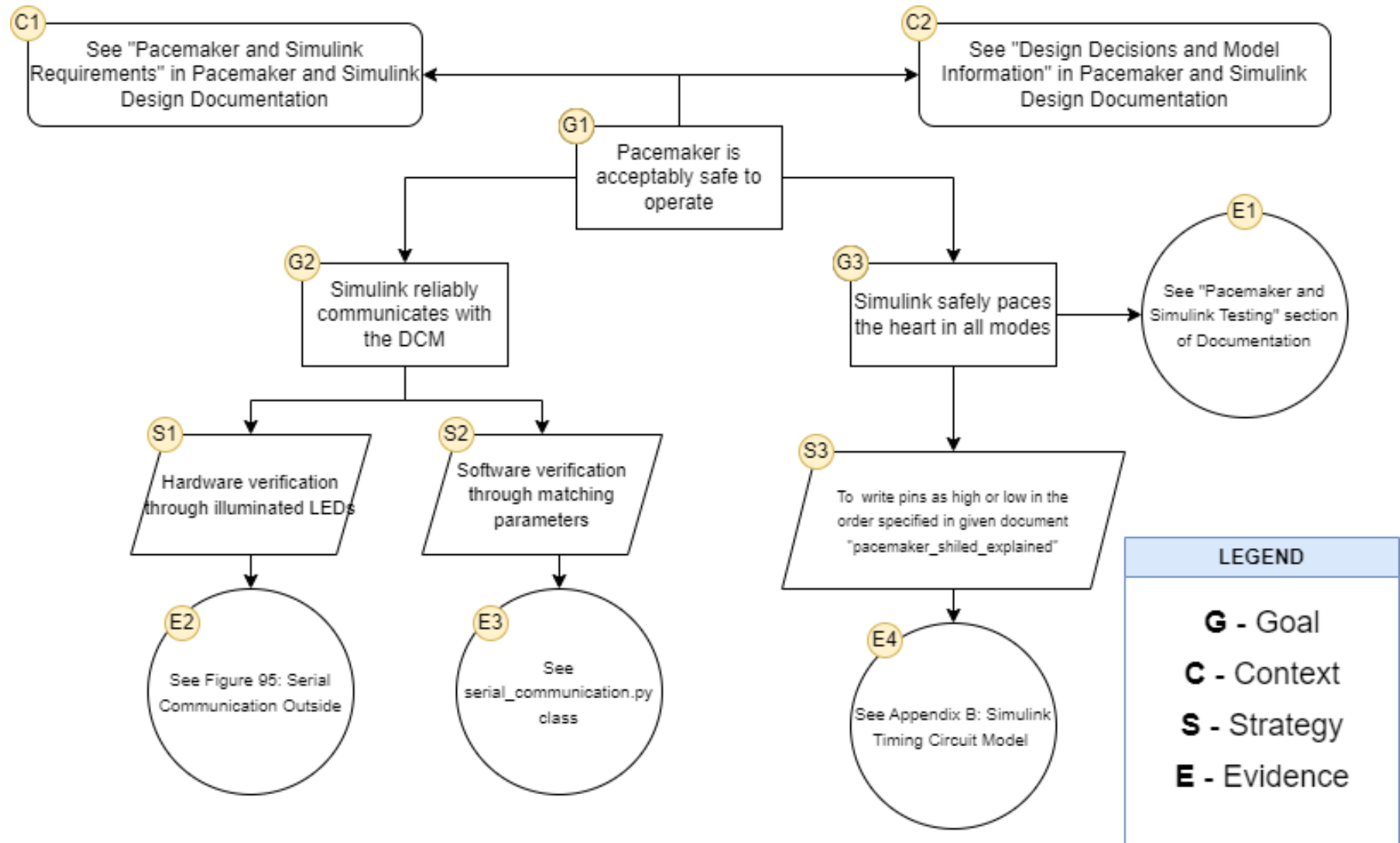
Table : the pacemaker and Heartview inputs for Test 3; testing the VVI mode at 60bpm while heart is at 59bpm.	41
Table : the pacemaker and Heartview inputs for Test 4; testing the VVI mode at 60bpm while heart is at 61bpm.	42
Table : the pacemaker and Heartview inputs for Test 5; testing the VVI mode at 60bpm while heart is at 120bpm.	43
Table : the pacemaker and Heartview inputs for Test 6; testing the VVI mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.	44
Table : the pacemaker and Heartview inputs for Test 7; testing the VVI mode at 60bpm while heart is at 30bpm and a target voltage of 10V.	45
Table : the pacemaker and Heartview inputs for Test 8; testing the VVI mode at 60bpm with an VRP of 150ms while heart is at 30bpm.	46
Table : the pacemaker and Heartview inputs for Test 9; testing the VVI mode at 60bpm with an ARP of 500ms while heart is at 30bpm.	47
Table : the pacemaker and Heartview inputs for Test 1; testing the AOOR mode with a LRL of 60bpm and URL of 120bpm.	48
Table : the pacemaker and Heartview inputs for Test 1; testing the VOOR mode with a LRL of 60bpm and URL of 120bpm..	49
Table 35: the pacemaker and Heartview inputs for Test 1; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.	50
Table 36: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.	51
Table 37: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.	52
Table 38: Table 37: the pacemaker and Heartview inputs for Test 1; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.	53
Table 39: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.	54
Table 40: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.	55
Table 41: DCM Requirements	57

Pacemaker and Simulink Design Documentation

Pacemaker and Simulink Requirements

Requirement	Description
1	<p>Implementation of the modes AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, VVIR.</p> <ul style="list-style-type: none">- AOO should pulse the atria at a desired threshold voltage and bpm.- VOO should pulse the ventricles at a desired threshold voltage and bpm.- AAI should pulse the atria at a desired threshold voltage and bpm if a natural atria pulse is not sensed by the pacemaker.- VVI should pulse the ventricles at a desired threshold voltage and bpm if a natural ventricle pulse is not sensed by the pacemaker.- AOOR should pulse the atria at a desired threshold voltage and bpm while also adapting the rate to accommodate accelerometer activity.- AOOR should pulse the atria at a desired threshold voltage and bpm while also adapting the rate to accommodate accelerometer activity.- VOOR should pulse the ventricles at a desired threshold voltage and bpm while also adapting the rate to accommodate accelerometer activity.- AAIR should pulse the atria at a desired threshold voltage and bpm if a natural atria pulse is not sensed by the pacemaker while also adapting the rate to accommodate accelerometer activity.- VVIR should pulse the ventricles at a desired threshold voltage and bpm if a natural ventricle pulse is not sensed by the pacemaker while also adapting the rate to accommodate for accelerometer activity.
2	<p>Values including mode, refractory period, atria threshold voltage, ventricular threshold voltage, pacemaker output voltage, target bpm, and pace duration can be manually customized by users through the user interface.</p>
3	<p>All modes should pass tests that adhere to the parameters outlined in Table 1.</p>
4	<p>The output of the Simulink model should be visible on Heartview and the user interface when dispatching tests.</p>
5	<p>Implement serial communication between the DCM and Simulink model</p>

Pacemaker and Simulink Safety Assurance Case Diagram:



Design Decisions and Model Information

Pacemaker Modes

The pacemaker and Simulink model currently has four different modes, all for treating Bradycardia, they include: AOO, VOO, AAI, and VVI. Every mode was assigned an integer so integer comparison could be used in conditional statements to execute certain blocks of code and state flows in the Simulink model. A description of the mode, it's parameters and paired integers are listed below:

Parameter	AOO (R)	VOO (R)	AAI (R)	VVI (R)			
Corresponding Integer	1	2	3	4	Increment	Nominal	Tolerance
Upper Limit	50-175ppm	50-175ppm	50-175ppm	50-175ppm	5ppm	120ppm	8ms
Lower Limit	30-50ppm, 50-90ppm, 90-175ppm	30-50ppm, 50-90ppm, 90-175ppm	30-50ppm, 50-90ppm, 90-175ppm	30-50ppm, 50-90ppm, 90-175ppm	5ppm, 1ppm, 5ppm	60ppm	8ms
Atrial Amplitude	Off,0.5-3.2V, 3.5-7.0V	NA	Off,0.5-3.2V (0.1), 3.5-7.0V (0.5)	NA	0.1V,0.5V	3.5V	12%
Atrial Pulse Width	0.05ms, 0.1-1.9ms	NA	0.05ms, 0.1-1.9ms	NA	0,0.1ms	0.4ms	0.2ms
Ventricular Amplitude	NA	Off,0.5-3.2V, 3.5-7.0V	NA	Off,0.5-3.2V, 3.5-7.0V	0.1V,0.5V	3.5V	12%
Ventricular Pulse Width	NA	0.05ms, 0.1-1.9ms	NA	0.05ms, 0.1-1.9ms	0:00	3.5V	12%
Atrial Sensitivity	NA	NA	(0.25,0.5,0.75), 1.0-10mV	NA	0, 0.5mV	0.75mV	20%
ARP	NA	NA	150ms-500ms (increment 10ms)	NA	10ms	250ms	8ms
PVARP	NA	NA	150-500ms	150-500ms	10ms	250ms	8ms
Ventricular Sensitivity	NA	NA	NA	(0.25,0.5,0.75), 1.0-10mV	0, 0.5mV	2.5mV	20%
VRP	NA	NA	NA	150-500ms	10ms	320ms	8ms
Rate Smoothing	NA	NA	0,3,6,9,12,15,18, 21,25%	0,3,6,9,12,15,18, 21,25%	NA	Off	1%

Table 1: the parameters and corresponding integers for the modes included in the Simulink model (AOO, VOO, AAI, and VVI)

AOO – will pace the atria of the heart at a selected voltage and bpm.

VOO – will pace the ventricles of the heart at a selected voltage and bpm.

AAI – will sense the atria and only pace them if it does not sense a natural pulse.

VVI – will sense the ventricles and only pace them if it does not sense a natural pulse.

AOOR – performs the same way as AOO with the added feature of changing the rate based on activity.

VOOR – performs the same way as VOO with the added feature of changing the rate based on activity.

AAIR – performs the same way as AAI with the added feature of changing the rate based on activity.

VVIR – performs the same way as VVI with the added feature of changing the rate based on activity.

Sensing Circuit Design

- See Appendix A for Simulink Model

The sensing circuit in the pacemaker is a key part of the AAI and VVI pacing modes. It is responsible for detecting natural heartbeats relaying that information into the timing chart. The sensing circuit is also responsible for taking in the input from the button and relaying that to the timing chart as well.

Input and Output Processes

The inputs into the timing chart are all user control constants. These include atrial threshold voltage, ventricular threshold voltage, the status of the circuit being on or off. We have implemented all of these as integer constant inputs into our circuit.

Input	Corresponding pin/name	Output	Corresponding pin/name
atrialThreshold	D3 – VENT_CMP_REF_PWM	buttonPushed	NA
ventThreshold	D6 – ATR_CMP_REF_PWM	atriumSensed	D0 – ATR_CMP_DETECT
status	D13 – FRONTEND_CTRL	ventricleSensed	D1 – VENT_CMP_DETECT

Table 2: Input and output corresponding pin names

The status of the circuit is simply one or zero which correlates to high and low voltages. This constant is inputted into the D13 pin which is the FRONTEND_CTRL. When this pin is set to high, the sensing circuitry is active. When it is set to low the sensing circuit is inactive. The other two inputs are the atrial and ventricular threshold voltages. The atrial and ventricular threshold voltages have the same application, except one is for the AAI mode and one for the VVI mode. These threshold voltages determine what the pacemaker considers a full heartbeat. If the heartbeat is strong enough that the voltage is above the threshold voltage, it won't assist with a manufactured pulse. If the natural beat is not detected, meaning the heartbeat was not strong enough, the pacemaker will assist and produce a pulse. This is done using multiple pins and some simple math. The D3 (VENT_CMP_REF_PWM) and D6 (ATR_CMP_REF_PWM) pins are connected to C22 capacitor which can be charged to specific voltage values between 0-5V. In Simulink, these pins accept values between 0 and 100 which correlates to the percentage of the 5V, I.e input of 60 charges the capacitor to 5×0.6 which is 3 volts. To convert our input threshold voltages into a percentage, we've used the product block in Simulink. This allows us to perform multiplication and division on multiple inputs into the block. We input our threshold voltage, the maximum voltage (5V) and the integer value of 100. The threshold voltage is multiplied by 100 and then

divided by 5 to get our final value to input into the D3 and D6 pins. We have these pins as PWM outputs. The next part of the sensing circuit uses the D0 (ATR_CMP_DETECT) and D1 (VENT_CMP_DETECT) pins which are also functionally the same but for the AAI and VVI modes respectively. These pins compare the sensing circuitry in the atrium/ventricle to the voltage that the D3 or D6 pin is set at. This means that when we use the above process to set the D6/D3 pins to our threshold voltage, it is the D0 and D1 pins which detect the natural beat voltage and output either a high voltage when the threshold is met, or a low voltage when the threshold isn't met. These are the 2 of the three outputs of the sensing circuit. We take these voltages from the D0 and D1 pins and input them into our timing circuit so that we can tell when to go into specific states. The last output is the button. This is simply a premade block in Simulink which outputs a high voltage when the button is pushed or a low voltage when it isn't being pushed. This is sent into the timing chart and used to hold the pacemaker in the recharge state. All three inputs into the sensing circuit should eventually be modifiable through the UI. For now, we have the status set as 1 for testing purposes to have it on. We have set the atrial and threshold voltages both as 3 volts. In a real-world application, this value is determined on a patient-by-patient basis. A threshold voltage that is too low means that the pacemaker won't emit a pulse when it needs to. A threshold voltage that is too high will lead to the pacemaker emitting a pace when the heart has already produced a sufficient natural pulse. What is considered a sufficient pulse is dependent on the individual patient. For our case we decided 3 is a good place to begin as we want to limit over and under sensing.

Timing Chart Design

- See Appendix B for Simulink Model

The timing chart has two main responsibilities. The first is determining which pacing mode to activate, and the second is the timing of the charging and discharging of capacitor C22 as well as the pin arrangement for each pacing mode.

Inputs and Outputs

The inputs that are required for the timing chart include pulseDuration, mode, refractoryPeriod, BPM, buttonPushed, atriumSensed, and ventricleSensed. The first four inputs listed are programmable parameters that in future designs will be controlled via the DCM, but currently they are implemented as constants. The mode input is what dictates which pacing mode the pacemaker will be set to, and the pulseDuration, refractoryPeriod and BPM inputs change the nature of the signal being sent to the patient's heart. The latter three inputs are from the sensing circuit and are set to either high or low, determined by if the natural atrium or ventricle pulses are being sensed, and if the push button on the pacemaker has been pressed. The timing chart will output eight values to the pacing circuit, which correspond to pins on the pacemaker that will create the signal required for pacing. The eight variables and their corresponding pins can be seen in the table below.

Variable	Pacemaker Pin	Variable	Pacemaker Pin
atrPaceControl	ATR_PACE_CTRL	paceChargeControl	PACE_CHARGE_CTRL
ventPaceControl	VENT_PACE_CTRL	paceGndControl	PACE_GND_CTRL
atrGndControl	ATR_GND_CTRL	zAtrControl	Z_ATR_CTRL
ventGndControl	VENT_GND_CTRL	zVentControl	Z_VENT_CTRL

Table 3: Timing chart variables with corresponding pacemaker variables

Timing Chart States

Once the inputs are set to their required values, the timing chart enters the initial state, denoted as State_Check. Upon entry, State_Check will define two values which will be used in the various pacing

modes. The first value, waitTime, is calculated using the equation $60000 / \text{BPM}$, 60000 being the number of milliseconds in one minute and BPM being the programmable input parameter. This value represents the time interval in milliseconds between each pulse sent to the heart. The second value, refractoryTime, is assigned the value from the input parameter refractoryPeriod, and represents the time interval in milliseconds following a natural event from the heart in which pacing cannot occur. Once these values are defined, the timing circuit will then enter the sub-chart that corresponds to the mode set by the user, thus starting the pacing process. Below is a table which will show what variables within the sub-charts correspond to the pins on the pacemaker.

For each pacing mode, the sub-chart can be thought of as three different states. Firstly, before any pacing can occur, the capacitor C22 must be charged, this can be thought of as state 1. This is done by setting the variable paceChargeControl to high, however, before that happens, atrPaceControl and ventPaceControl must be set to low to protect the patient from the charging PWM signal. Once the capacitor is charged, the pulse can then be delivered, this can be thought of as state 2. This is done by first setting paceChargeControl to low, and then setting paceGndControl and atrPaceControl or ventPaceControl to high. The decision between atrPaceControl and ventPaceControl is determined by the mode the pacemaker is in. Now that the pulse has been delivered, the timing chart will enter state 3, which is responsible for charging the C21 blocking capacitor and to offset the charge built up by the pacing current. Firstly, atrPaceControl or ventPaceControl must be set to low, and then atrGndControl or ventGndControl will be set to high. Once again, the decision between the two different pairs of variables is determined by the mode of the pacemaker. Once the charging of C21 is complete, the system will then return to state 1 and begin the cycle again. One optimization made was to combine states 1 and 3 to occur in parallel, since both states are independent from one another and perform a similar task, being the charging of a capacitor. With this optimization made, the variable layout for each state can be seen below. Note that the layout shown is for modes that involve the atrium, for modes that involve the ventricle, the only change required to make is to flip the Boolean value of the corresponding ventricle variable to high and set the atrium variable to low. Variable layouts for specific modes can be found in the appendix.

Variable	Boolean Value
atrPaceControl	False
ventPaceControl	False
paceChargeControl	True
paceGndControl	True
zAtrControl	False
zVentControl	False
atrGndControl	True
ventGndControl	False

Table 4: Variable layout for AOO and AAI charging state.

Variable	Boolean Value
paceChargeControl	False
paceGndControl	True
atrPaceControl	True
atrGndControl	False
zAtrControl	False
zVentControl	False
ventGndControl	False
ventPaceControl	False

Table 5: Variable layout for AOO and AAI pacing state

AOO and VOO Timing

The AOO and VOO pacing modes are functionally identical to each other, the only difference being that AOO paces the atrium and VOO paces the ventricle. As described above, once the timing chart has entered the corresponding sub-chart, it will first enter the charging state. Upon entry into the charging state, the variables are set to their respective Boolean values for charging and then a check is made to ensure that the input variable `buttonPushed` from the sensing circuit is set to `False`. This condition was made so that in the future once the pacing is controlled by the DCM, the user can inhibit pacing in case any erroneous inputs were made during configuration. After that check is made, it will then wait for the time interval defined in `waitTime`, before entering the pacing state. Upon entry into the pacing state, the variables are set to their respective Boolean values for pacing. This will alter the outputs of the timing chart and deliver the pacing signal to the patient via the pacing circuit. Before exiting the pacing state and restarting the cycle, it first waits for the interval of time defined in the input parameter `pulseDuration`, and then sets the `waitTime` value to be $(60000 / \text{BPM}) - \text{pulseDuration}$. This is done so that the total time to complete a cycle is equal to exactly $60000 / \text{BPM}$, since the time spent between each state is equal to `waitTime` + `pulseDuration`. Therefore, if the `waitTime` is left unaltered, the time spent between the states would be greater than $60000 / \text{BPM}$ and the pacing would be off. This new `waitTime` value is then used for the rest of the duration that the pacemaker is in that respective mode.

AAI and VVI Timing

Similarly, to AOO and VOO, AAI and VVI share many characteristics that lead to the implementations being similar to one another. Upon entry into the AAI and VVI sub-charts, the first state is once again the recharge state where output variables are set to charge the capacitors. The same check for `buttonPushed` is in place for both modes as well. After that, instead of waiting for the `waitTime` interval, it will then run a second check to see if the atrium or ventricle has been sensed. This is accomplished with a conditional statement that checks to see if the `atriumSensed` or `ventricleSensed` input variables are set to `True`, the choice between the two variables depends on which mode it is in. If so, then the program will wait for the time interval `refractoryTime`, set `refractoryTime` to equal the `refractoryPeriod` given in the input, and also set `waitTime` to be $60000/\text{BPM}$ before re-entering into the charging state. The two assignment operations performed are required to keep the time for each cycle constant, similarly to AOO and VOO after the pacing state we must alter the time intervals to adjust for the duration of the pulse. If the atrium or ventricle are not sensed, it will then enter the pacing state where the variables are set for pacing, it waits for the pulse duration, adjust the `waitTime` and `refractoryTime`, and then returns to the charging state.

Rate Adaptive Modes Timing: AOOR, VOOR, AAIR, and VVIR

The rate adaptive timing modes operate on the same timing circuit as their respective standard modes described above including the same input and output variables. The only difference is that the rate adaptive modes allow the pacemaker to speed up and slow down pulses based on the activity sensed by the accelerometer on the hardware. This data is stored in the variable `Changing_LRL` in the Simulink model which is essentially the pacing rate of the pacemaker and is updated based on the sensed activity. If no activity is sensed the pacing rate of the pacemaker is just the input `LRL`. The activity is supposed to mimic if someone was wearing the pacemaker and exceeded their resting heart rate (lower rate limit) by walking, jogging, or running. Based on the sensed activity, the pacemaker will pace at a new rate that is calculated by the “Rate Adaptive Calculation” subsystem in Simulink. For example, if the mode AOOR is enabled and the hardware is shaken fast, the pacemaker will pace at a faster rate. Once the board is no

longer being shaken and the activity stops, the pacemaker will slowly return to the set resting heart rate (lower rate limit).

Pacemaker Pacing Design

- See Appendix C for Simulink Model

The pacing circuit is responsible for setting the pins on the pacemaker to match the output from the timing chart for pacing delivery. It takes the 8 outputs from the timing chart as inputs, and digital writes the Boolean value associated with each output to the corresponding pin. For example, for mode AOO during the pacing state, the pacing circuit will write high to the pins PACE_GND_CTRL, and ATR_PACE_CTRL, and write low to the other 6 pins. In addition to setting pins on pacemaker, the pacing circuit also generates a PWM output that represents the amplitude of the paced signal. This is done using the same technique from the sensing circuit, where the pins VENT_CMP_REF_PWM and ATR_CMP_REF_PWM were set. As a reminder, the PWM output takes a value from 0 to 100 which represents the percentage of the 5V capacity of capacitor C22, using the product block to multiply the wanted amplitude by 100 and then dividing by 5, a percentage is achieved which represents the desired amplitude. For example, if the target amplitude is 3.5V, the constant which currently represents pulse amplitude would be set to 3.5, $3.5 * 100 / 5 = 70\%$. In the future, pulse amplitude will not be represented as a constant and instead will be a programmable parameter from the DCM.

Rate Adaptive Calculation and Accelerometer Activity

Accelerometer Subsystem:

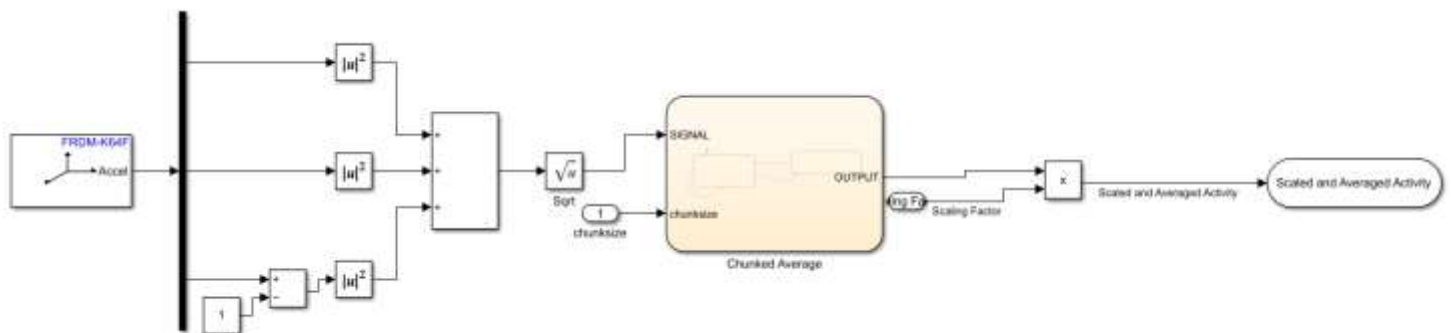


Figure 1: Accelerometer Subsystem in Simulink

The Accelerometer Subsystem takes the activity sensed by the accelerometer and converts it to a numerical value. The accelerometer senses movement on three axes: x-axis, y-axis, and z-axis. The Accelerometer Subsystem performs this conversion by taking the dot product of the acceleration vectors of each axis. As the accelerometer continues to experience motion the new acceleration vectors accumulate with the old ones to create rolling average. The average is then output as the accelerometer activity to best used in the rate adaptive calculation. Once the accelerometer stops experiencing motion the activity from the accelerometer returns to zero. If the accelerometer starts to experience motion again the model will begin to calculate a new rolling average and the process repeats.

Rate Adaptive Calculation:

For the rate adaptive modes, we need to change the rate based on the level of activity being sensed from the accelerometer. The way the activity is interpreted is explained above. The rate adaptive calculation sub system then receives a scaled average of what the accelerometer is sensing. The input that it receives that goes through a sub system is the activity threshold. The activity threshold is essentially how much activity we want to sense before we begin to increase the target BPM. A higher threshold allows for more motion before the rate increases. This threshold is received as a value between 0-6 from the serial communication system, and that value is then translated to a corresponding numerical threshold. The model and inputs are shown below.

Input	Origin Location
ACTIVITY	Accelerometer subsystem
LRL	Serial In
URL	Serial In
MSR	Serial In
A_THRESH	Threshold Conversion
REACTION_TIME	Serial In
RESPONSE_TIME	Serial In
MODE	Serial In

Table 6: Rate adaptive inputs

The rate calculation develops a linear slope based on the response factor, maximum sensing rate, and the activity threshold. The target rate is then calculated using the activity level received from the accelerometer, the activity threshold, and the lower rate limit. Once the target is calculated, It is checked to see if it is higher than the upper limit, and if it is lower than the lower limit. In these cases, the target rate is set to the respective limit. Next, the step is calculated to see how much we will change the rate. This uses the reaction time and recovery time which are received from serial communication. If the target rate is higher than the actual, we will add the step size calculated for reaction, and if the actual rate is lower than the target rate, we will subtract the step size calculated for recovery. This essentially implements hysteresis as the reaction time will always be greater than the recovery time. The response factor also allows for variable slopes which can change the speed at which the rate reacts. The different intended behaviors for the rates are shown below for both increasing and decreasing.

Decreasing:

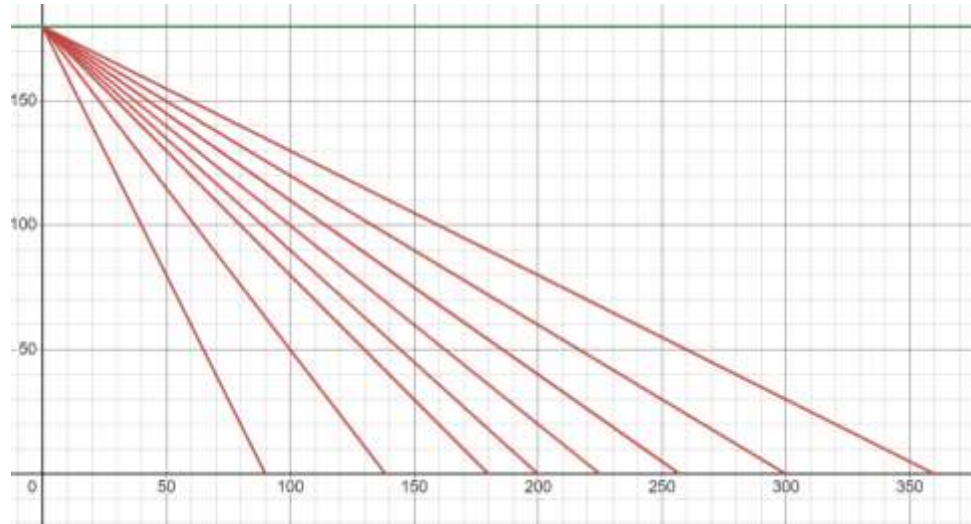


Figure 2: desired behavior for decreasing activity

Increasing:

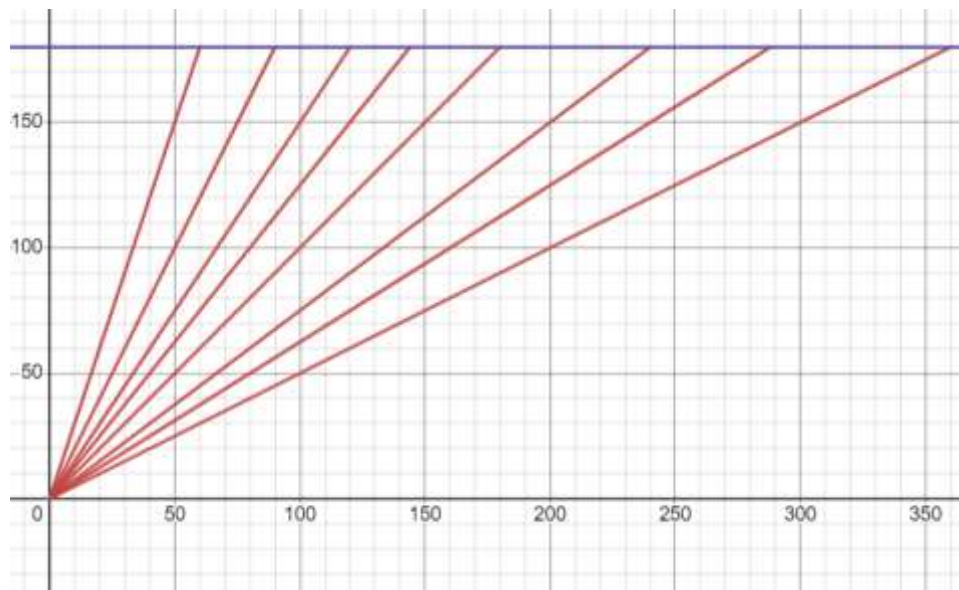


Figure 3: desired behavior for increasing activity

Between the response factor, reaction time, and recovery time, the design allows for full customization of how the rate changes based on each patient's needs.

Serial Communication

Serial Receive

Serial communication between the pacemaker and the DCM is achieved using the UART protocol. Within Simulink, a 40-byte array is expected to be received that contains all the necessary information for the pacemaker. Using the serial receive block, this data named rxdata is sent as an input to the serial receive chart. In addition, a status of 0 or 1 is outputted from the block to signal transmission which is also configured as an input. Below is a table that shows all the information expected, along with the size of each input parameter.

Input Parameter	Data Type	Size in Bytes
Transmission Byte	UInt8	1
Decision Byte	UInt8	1
MODE	UInt8	1
LRL	UInt16	2
URL	UInt16	2
MSR	UInt16	2
A_AMPLITUDE	Single	4
V_AMPLITUDE	Single	4
A_WIDTH	UInt16	2
V_WIDTH	UInt16	2
A_SENSITIVITY	Single	4
V_SENSITIVITY	Single	4
VRP	UInt16	2
ARP	UInt16	2
ACTIVITY_THRESH	UInt8	1
REACT_TIME	UInt16	2
RESPONSE_FAC	UInt16	2
RECOVERY_TIME	UInt16	2

Table 7: Input parameter datatypes

Upon initialization of the model (flash to the board), nominal values are set for each parameter to prevent errors from occurring within the model. Additionally, it provides a safety feature to ensure that the pacemaker is working correctly before any values are sent from the DCM. Below is a table that shows the initial values of each parameter.

Parameter	Nominal Value
MODE	1 (Integer that corresponds with mode VOO)
LRL	60
URL	120
MSR	120
A_AMPLITUDE	5
V_AMPLITUDE	5
A_WIDTH	1
V_WIDTH	1
A_SENSITIVITY	0
V_SENSITIVITY	0
VRP	320

ARP	250
ACTIVITY_THRESH	3 (Integer that corresponds with mode Medium)
REACT_TIME	30
RESPONSE_FAC	8
RECOVERY_TIME	5

Table 8: Input parameter nominal values

Once initialization is completed, the model will enter the STANDBY state where it will wait indefinitely until a transmission is received from the DCM. Within this STANDBY state, parameters, RED_LED, GREEN_LED, BLUE_LED are set to 0 to show that there is no longer any data being received. The start of transmission is signaled by the transmission byte being received as 0. This is checked with the statement `rxdata(1) == hexdec('00')`. The DCM sends the uint8 as a hexadecimal number so conversion is required to perform the equals operation. The following byte, called the decision byte, is sent as either a 1 or 2. This byte is responsible for deciding whether to update pacemaker parameters, or to send data for an egram. If the decision byte is a 1, the model will enter into the UPDATE_PARAMETERS state, where the programmable parameters are updated to match the values sent by the DCM. This is done by indexing the rxdata array with the correct number of byte at the corresponding position for each parameter. For example, to receive the A_AMPLITUDE parameter, the command would look like `A_AMPLITUDE = typecast(rxdata(10:13), 'single')`. A_AMPLITUDE is a single so 4 bytes are required to hold the information, and those 4 bytes are at the positions 10, 11, 12, and 13 of the array. A typecast is required to consolidate the 4 bytes into one value within Simulink. This same process is done for each parameter sent, after which the RED_LED is set to 1 to signal that the data has been received. Before returning to the STANDBY state, an intermediary state called SEND_PARAMETERS is entered to send the newly updated parameters back to the DCM for confirmation that they are all correct. Once this is done, GREEN_LED is set to 1 to signal that the parameters have been sent back and then STANDBY is entered once again. If instead, the decision byte is sent as a 2, the model will enter the SEND_EGRAM state where similarly to SEND_PARAMETERS, the current parameters are sent back to the DCM, but this time for the purposes of the egram instead of confirmation. Each time the SEND_EGRAM state is entered, the BLUE_LED is set to 1 to signal the parameters have been sent. Once again after this action is completed the model will enter the STANDBY state. Each of the parameters sent by the DCM except for the transmission byte and the decision byte are configured as outputs of the chart and are used throughout the entire rest of the system.

Serial Transmit

Each output from the serial receive chart is sent as an input into the serial transmit subsystem. This subsystem is a function call subsystem with the name SEND_PARAMETERS and it is called every time the DCM sends parameters to be updated, or requests data for an egram. Every input larger than a byte is then sent into a byte-packer, that splits the larger data type into multiple bytes, so that it can be sent back through the serial transmit block. The byte packer is configured to be the same size as the datatype being sent through it, so that there are no empty bytes sent. All packed bytes are then sent into a multiplexer, which is connected to the serial transmit block. In addition to the input data being sent back to the DCM, analog data from the pins A0 and A1 is sent for the egram. This data is of type double and is sent to the DCM regardless of if an egram is requested or not because the data sent through the serial transmit block must be an array of consistent size. Refer to appendix E to see all aspects of the serial communication Simulink model.

Pacemaker and Simulink Testing

Mode	Test	Expected Result	Result	Pass/Fail
AOO	1	Atrium pacing at 60 bpm	<u>1</u>	Pass
	2	Atrium pacing at 120 bpm	<u>2</u>	Pass
	3	No pacing	<u>3</u>	Pass
	4	No pacing	<u>4</u>	Pass
VOO	1	Ventricle pacing at 60 bpm	<u>1</u>	Pass
	2	Ventricle pacing at 30 bpm	<u>2</u>	Pass
	3	No pacing	<u>3</u>	Pass
	4	No pacing	<u>4</u>	Pass
AAI	1	Atrium pacing at 60 bpm	<u>1</u>	Pass
	2	No pacing	<u>2</u>	Pass
	3	Atrium pacing at 59 bpm	<u>3</u>	Pass
	4	No pacing	<u>4</u>	Pass
	5	No pacing	<u>5</u>	Pass
	6	No pacing	<u>6</u>	Pass
	7	No pacing	<u>7</u>	Pass
	8	Pacing at 60 bpm	<u>8</u>	Pass
	9	Pacing at 60 bpm	<u>9</u>	Pass
VVI	1	Atrium pacing at 60 bpm	<u>1</u>	Pass
	2	No pacing	<u>2</u>	Pass
	3	Atrium pacing at 59 bpm	<u>3</u>	Pass
	4	No pacing	<u>4</u>	Pass
	5	No pacing	<u>5</u>	Pass
	6	No pacing	<u>6</u>	Pass
	7	No pacing	<u>7</u>	Pass
	8	Pacing at 60 bpm	<u>8</u>	Pass
	9	Pacing at 60 bpm	<u>9</u>	Pass
AOOR	1	Increased atrium pacing when there is activity	<u>1</u>	Pass
VOOR	1	Increased ventricle pacing when there is activity	<u>1</u>	Pass
AAIR	1	No pacing when there is no activity and atrium pacing when there is activity	<u>1</u>	Pass
	2	Atrium always pacing	<u>2</u>	Pass
	3	Atrium never pacing	<u>3</u>	Pass
VVIR	1	No pacing when there is no activity and ventricle pacing when there is activity	<u>1</u>	Pass
	2	Ventricle always pacing	<u>2</u>	Pass
	3	Ventricle never pacing	<u>3</u>	Pass

AOO Testing

Test 1: Displaying AOO Bradycardia Therapy at 60bpm and a target voltage of 3V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 9: the pacemaker and Heartview inputs for Test 1; testing the AOO mode at 60bpm and a target voltage of 3V.

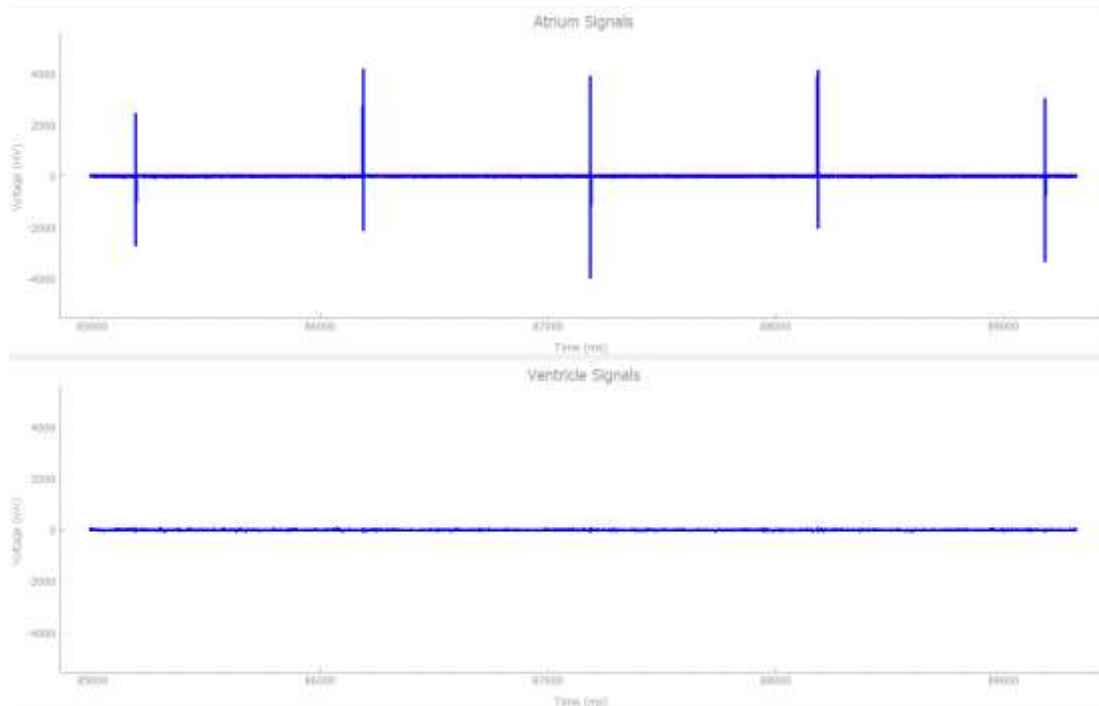


Figure 4: the Heartview output for Test 1; testing the AOO mode at 60bpm and a target voltage of 3V.

Figure 4 displays the outputs of AOO Test 1. This is the output we would have expected for testing the AOO mode given the input values into the model and Heartview. The plot only displays the generated pulses by the pacemaker to the atria which are equal to ~3V at a pace of about 1 beat per 1000ms which is equivalent to 60bpm.

Test 2: Displaying AOO Bradycardia Therapy at 120bpm and a target voltage of 1V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	120	Natural Ventricle	OFF
Target Output Voltage (V)	1	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 10: the pacemaker and Heartview inputs for Test 2; testing the AOO mode at 120bpm and a target voltage of 1V.

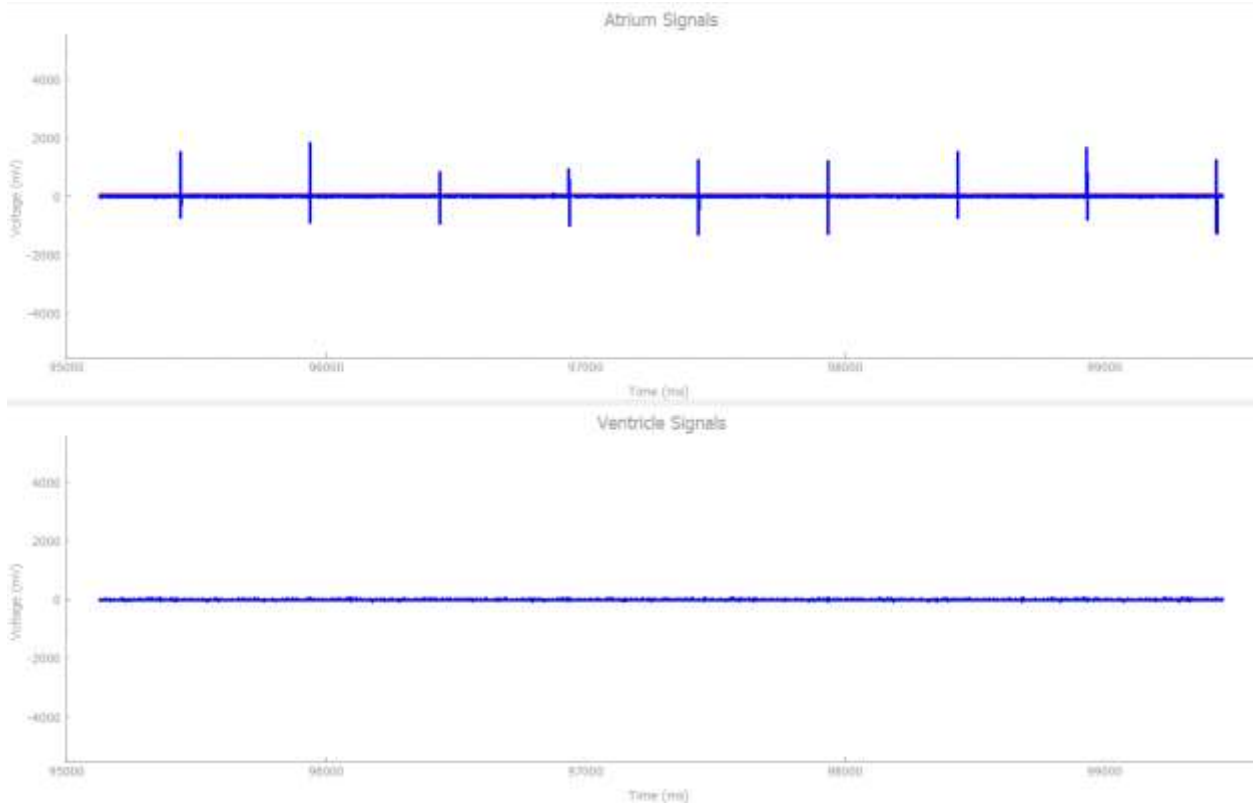


Figure 5: the Heartview output for Test 2; testing the AOO mode at 120bpm and a target voltage of 1V.

Figure 5 displays the outputs of AOO Test 2. This is the output we would have expected for Test 2 of the AOO mode given the input values into the model and Heartview. The plot only displays the generated pulses by the pacemaker to the atria which are equal to ~1V at a pace of about 2 beats per 1000ms which is equivalent to 120bpm. Using Test 1 as baseline, the pulses for Test 2 are much smaller than Test 1 with respect to voltage and happen twice as often is expected.

Test 3: Testing AOO Bradycardia Therapy at 60bpm and a target voltage of 10V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	10	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 11: the pacemaker and Heartview inputs for Test 3; testing the AOO mode at 60bpm and a target voltage of 10V.

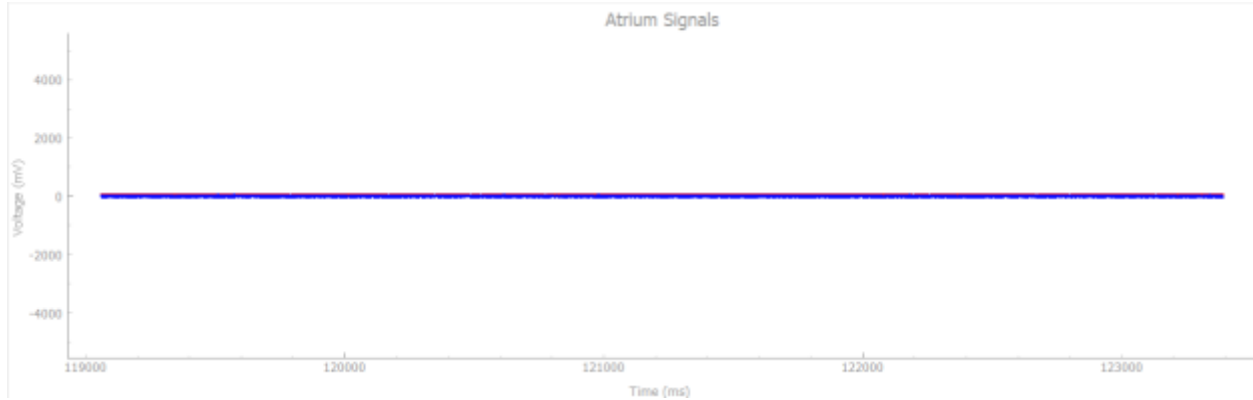


Figure 6: the Heartview output for Test 3; testing the AOO mode at 60bpm and a target voltage of 10V.

Figure 6 displays the outputs of AOO Test 3. This is the output we would have expected for Test 3 of the AOO mode given the input values into the model and Heartview. The plot displays no values because the input target voltage (10V) exceeds the threshold voltage (5V). This is because the capacitor C22 can only store a voltage of 5V. This means that we have designed our model so that the pacemaker only paces the atria at voltages of 5V or less.

Test 4: Testing AOO Bradycardia Therapy at 60bpm and a target voltage of 5V when the button SW2 is pushed.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	5	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 12: the pacemaker and Heartview inputs for Test 4; testing the AOO mode at 60bpm and a target voltage of 5V when the button is switched off.

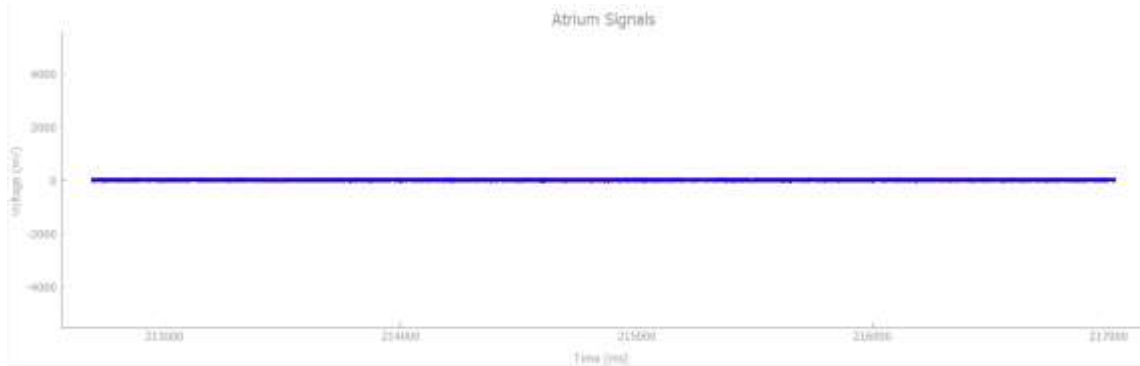


Figure 7: the Heartview output for Test 4; testing the AOO mode when button SW2 is pressed.

Figure 7 displays the outputs of AOO Test 4. This is the output we would have expected for Test 4 of the AOO mode given the input values into the model and Heartview while button SW2 is being pushed. The plot displays no values because we have designed our code such that when button SW2 is pressed the AOO mode is “stuck” in a state where it is charging the capacitor C21 and will not pace the atria.

VOO Testing

Test 1: Displaying VOO Bradycardia Therapy at 60bpm and a target voltage of 3V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 13: the pacemaker and Heartview inputs for Test 1; testing the VOO mode at 60bpm and a target voltage of 3V.

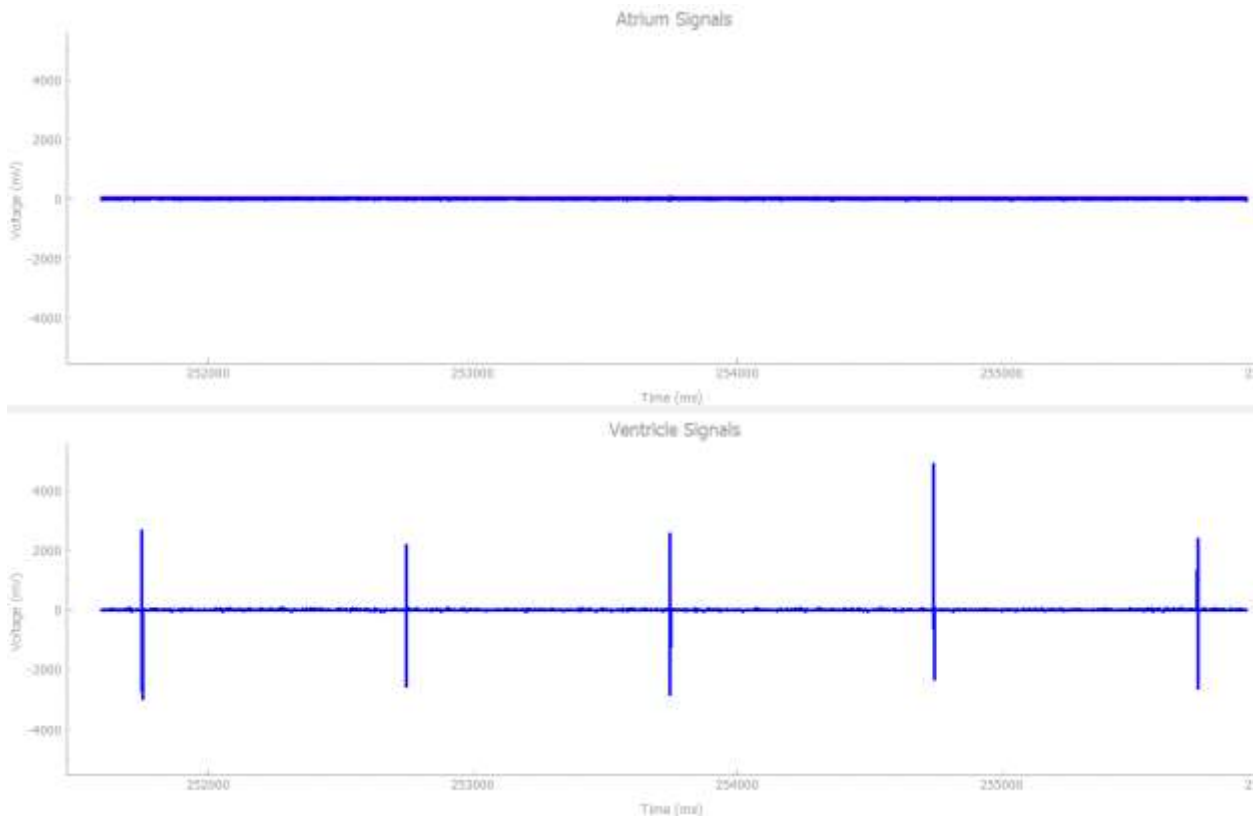


Figure 8: the Heartview output for Test 1; testing the VOO mode at 60bpm and a target voltage of 3V.

Figure 8 displays the outputs of VOO Test 1. This is the output we would have expected for testing the VOO mode given the input values into the model and Heartview. The plot only displays the generated pulses by the pacemaker to the ventricles which are equal to ~3V at a pace of about 1 beat per 1000ms which is equivalent to 60bpm.

Test 2: Displaying VOO Bradycardia Therapy at 30bpm and a target voltage of 5V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	30	Natural Ventricle	OFF
Target Output Voltage (V)	5	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 14: the pacemaker and Heartview inputs for Test 2; testing the AOO mode at 30bpm and a target voltage of 5V.

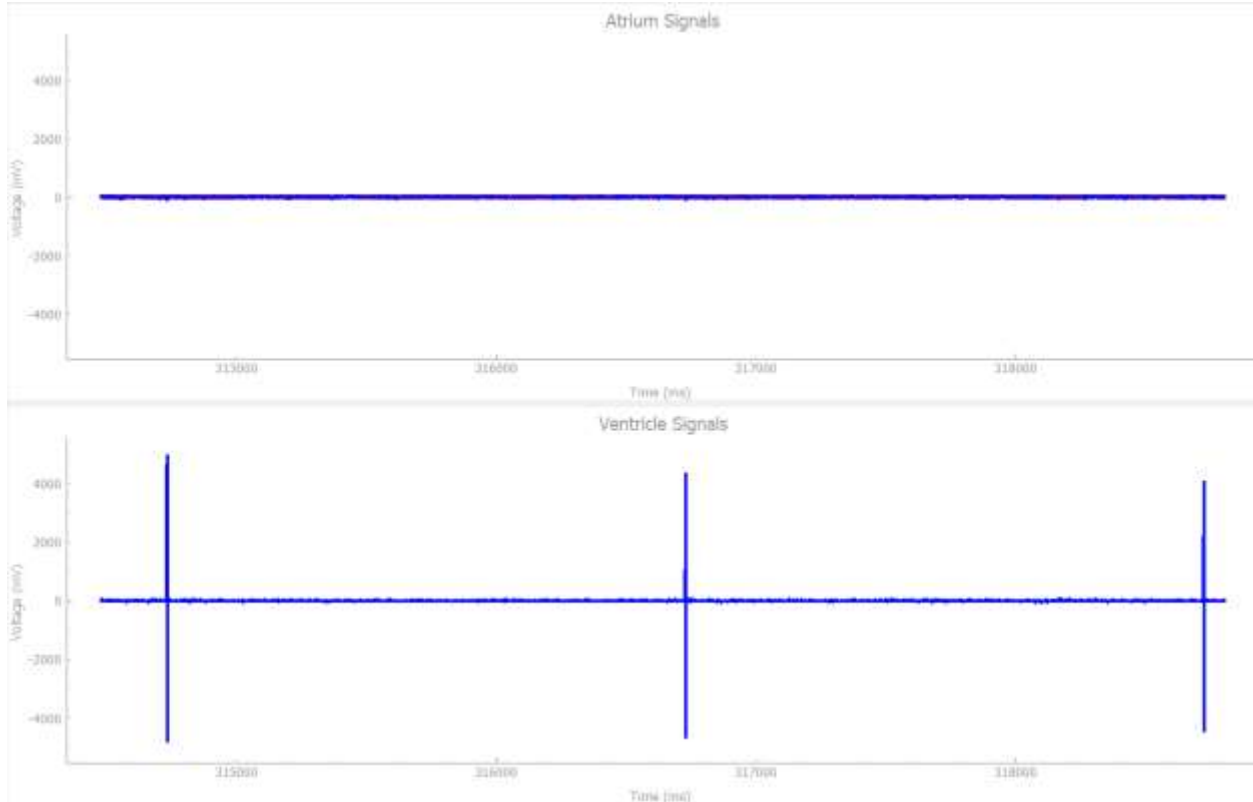


Figure 9: the Heartview output for Test 2; testing the VOO mode at 30bpm and a target voltage of 5V.

Figure 9 displays the outputs of VOO Test 2. This is the output we would have expected for Test 2 of the VOO mode given the input values into the model and Heartview. The plot only displays the generated pulses by the pacemaker to the atria which are equal to ~5V at a pace of about 1 beat per 2000ms which is equivalent to 30bpm. Using VOO Test 1 as baseline, the pulses for Test 2 are larger than Test 1 with respect to voltage and happen at half the pace as often is expected.

Test 3: Testing VOO Bradycardia Therapy at 60bpm and a target voltage of 10V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	15	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 15: the pacemaker and Heartview inputs for Test 3; testing the VOO mode at 60bpm and a target voltage of 10V.

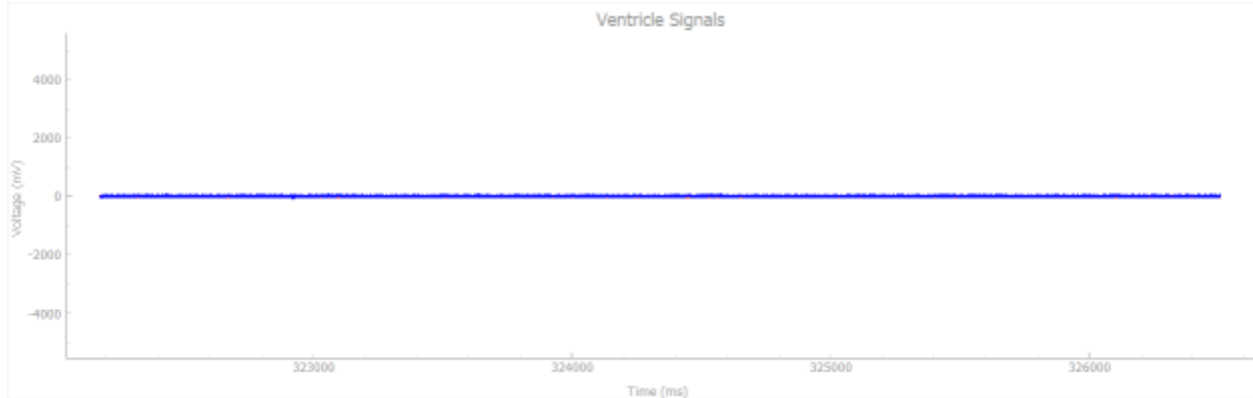


Figure 10: the Heartview output for Test 3; testing the VOO mode at 60bpm and a target voltage of 15V.

Figure 10 displays the outputs of VOO Test 3. This is the output we would have expected for Test 3 of the VOO mode given the input values into the model and Heartview. The plot displays no values because the input target voltage (15V) exceeds the threshold voltage (5V). This is because the capacitor C22 can only store a voltage of 5V. This means that we have designed our model so that the pacemaker only paces the ventricles at voltages of 5V or less.

Test 4: Testing AOO Bradycardia Therapy at 60bpm and a target voltage of 5V when the button SW2 is pushed.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	1	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	5	Natural Heart Rate	N/A
Pulse Duration (s)	1	Natural AV Delay	N/A

Table 16: the pacemaker and Heartview inputs for Test 4; testing the VOO mode at 60bpm and a target voltage of 5V when the button is pushed.

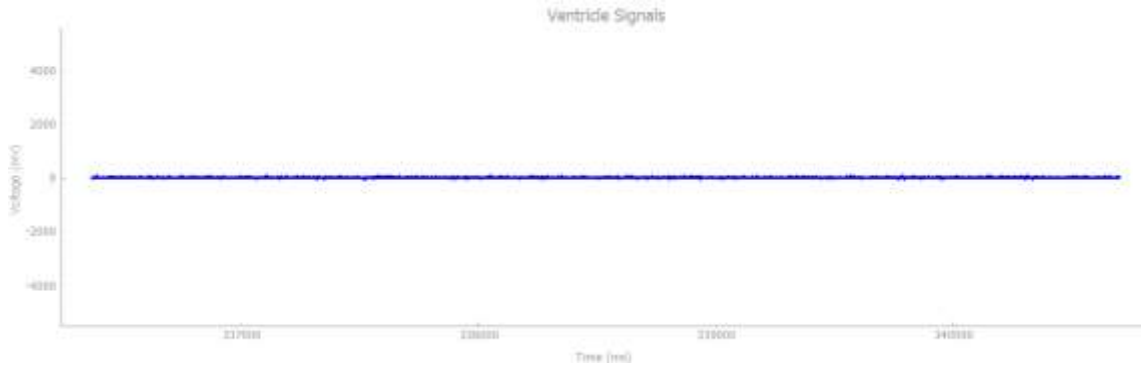


Figure 11: the Heartview output for Test 4; testing the VOO mode when button SW2 is pressed.

Figure 11 displays the outputs of VOO Test 4. This is the output we would have expected for Test 4 of the VOO mode given the input values into the model and Heartview while button SW2 is being pushed. The plot displays no values because we have designed our code such that when button SW2 is pressed the VOO mode is “stuck” in a state where it is charging the capacitor C21 and will not pace the ventricles.

AAI Testing

- The refractory period was set to 250ms for AAI testing because that is the value which was specified for Atrial Refractory Period in Table 1.

Test 1: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 30bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 17: the pacemaker and Heartview inputs for Test 1; testing the AAI mode at 60bpm while heart is at 30bpm.

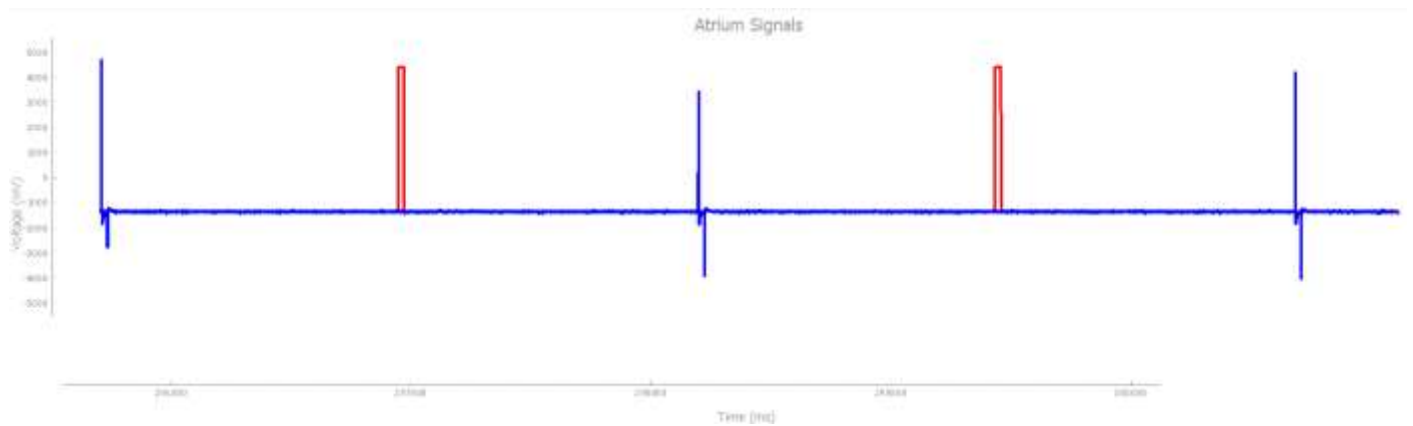


Figure 12: the Heartview output for Test 1; testing the AAI mode at 60bpm while heart is at 30bpm.

Figure 12 displays the outputs of AAI Test 1. This is the output we would have expected for Test 1 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 30bpm and blue pulses (pacemaker) paced at 60bpm. The target bpm is 60bpm however the heart is only pacing at 30bpm. Therefore, the pacemaker needs to pace the atria in between natural heart beats to compensate for the slow natural heart rate.

Test 2: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 60bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	60
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 18: the pacemaker and Heartview inputs for Test 2; testing the AAI mode at 60bpm while heart is at 60bpm.

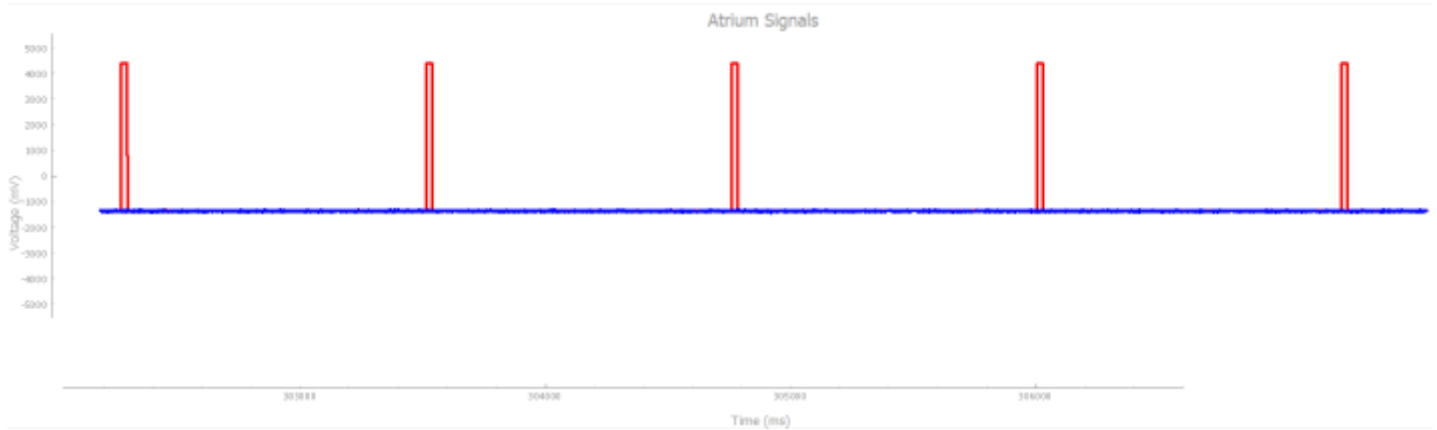


Figure 13: the Heartview output for Test 2; testing the AAI mode at 60bpm while heart is at 60bpm.

Figure 13 displays the outputs of AAI Test 2. This is the output we would have expected for Test 2 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). The blue pulses do not appear because the heart is naturally beating at the target bpm so there is no reason for the pacemaker to intervene.

Test 3: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 59bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	59
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 19: the pacemaker and Heartview inputs for Test 3; testing the AAI mode at 60bpm while heart is at 59bpm.

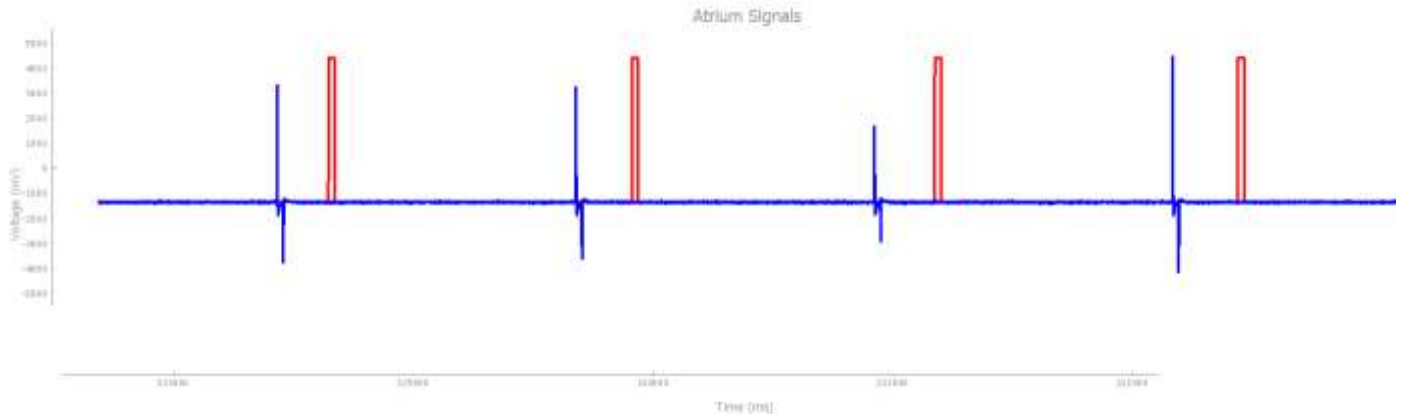


Figure 14: the Heartview output for Test 3; testing the AAI mode at 60bpm while heart is at 59bpm.

Figure 14 displays the outputs of AAI Test 3. This is the output we would have expected for Test 3 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and blue pulses (pacemaker). This is a boundary test case to ensure that the pacemaker is providing a pulse to the heart even when it is just shy of the target bpm (60bpm).

Test 4: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 61bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	61
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 20: the pacemaker and Heartview inputs for Test 4; testing the AAI mode at 60bpm while heart is at 61bpm.

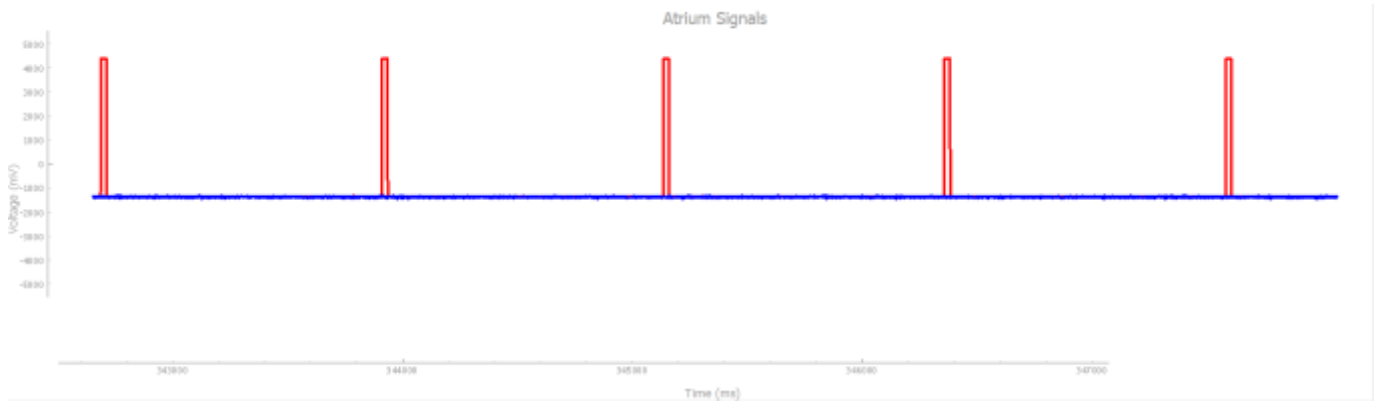


Figure 15: the Heartview output for Test 4; testing the AAI mode at 60bpm while heart is at 61bpm.

Figure 15 displays the outputs of AAI Test 4. This is the output we would have expected for Test 4 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is a boundary test case to ensure that the pacemaker is not providing a pulse to the heart even when it is just above the target bpm (60bpm).

Test 5: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 120bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	120
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 21: the pacemaker and Heartview inputs for Test 5; testing the AAI mode at 60bpm while heart is at 120bpm.

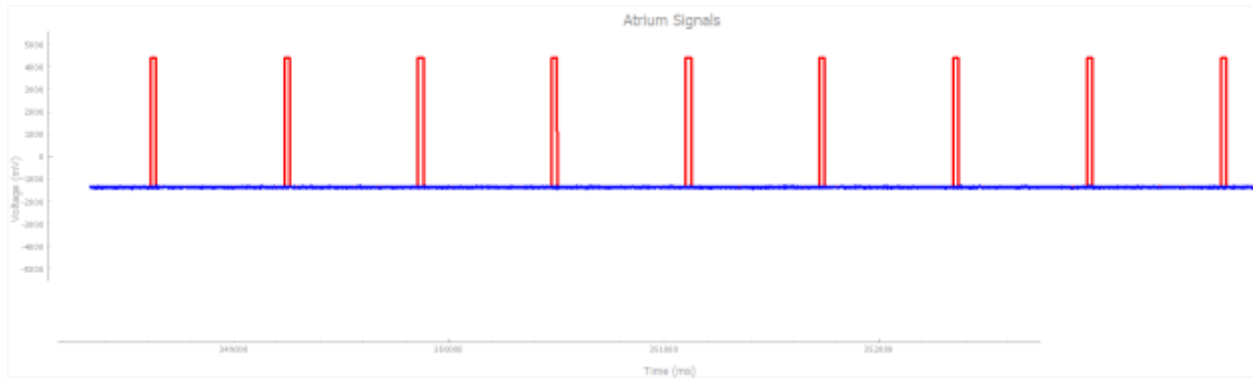


Figure 16: the Heartview output for Test 5; testing the AAI mode at 60bpm while heart is at 120bpm.

Figure 16 displays the outputs of AAI Test 5. This is the output we would have expected for Test 5 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is because the heart is already naturally pacing at above its target bpm.

Test 6: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 30bpm while button SW2 is pressed.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 22: the pacemaker and Heartview inputs for Test 6; testing the AAI mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.

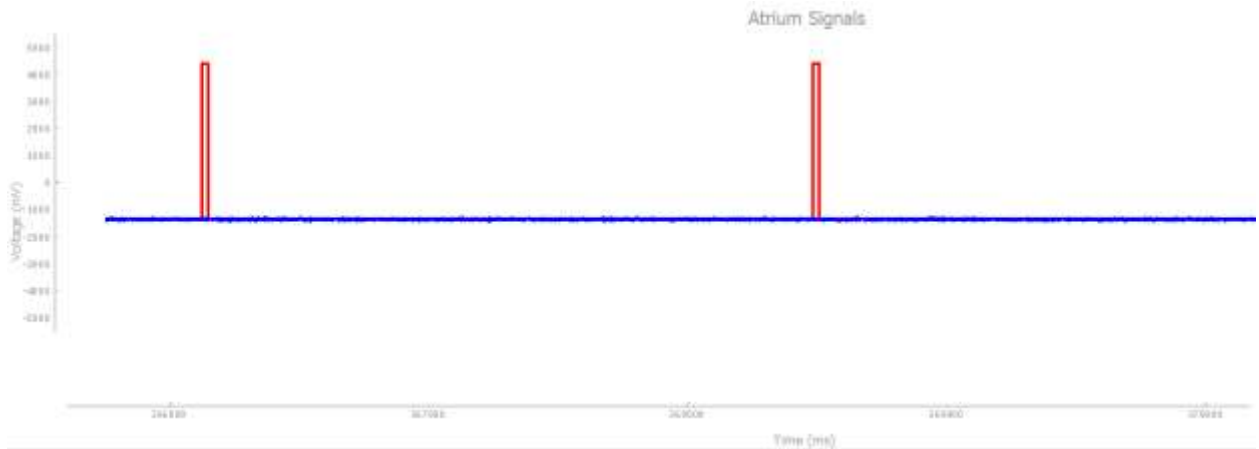


Figure 17: the Heartview output for Test 6; testing the AAI mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.

Figure 17 displays the outputs of AAI Test 6. This is the output we would have expected for Test 6 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is because the pulses from the pacemaker are inhibited due to the SW2 button being pressed as written in our Simulink model.

Test 7: Displaying AAI Bradycardia Therapy at 60bpm while heart is at 30bpm and a target voltage of 10V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	10	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	250

Table 23: the pacemaker and Heartview inputs for Test 7; testing the AAI mode at 60bpm while heart is at 30bpm and a target voltage of 10V.

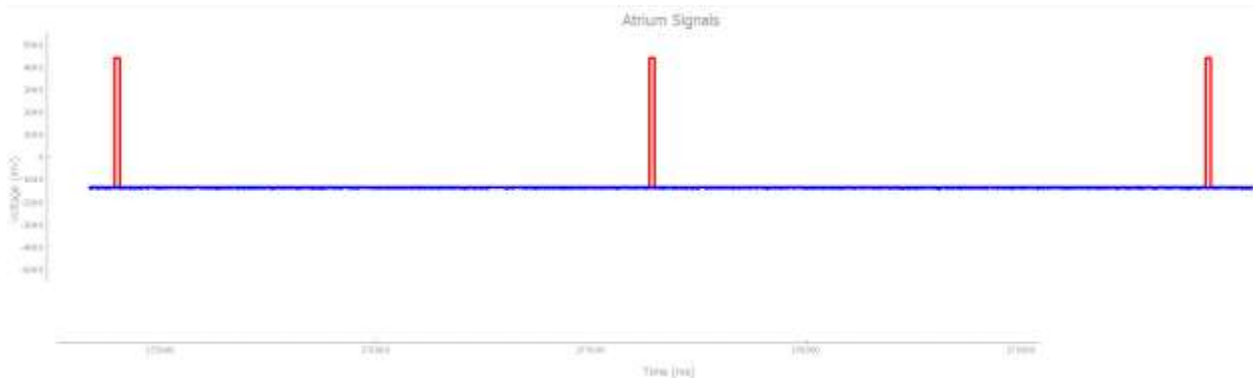


Figure 18: the Heartview output for Test 7; testing the AAI mode at 60bpm while heart is at 30bpm and a target voltage of 10V.

Figure 18 displays the outputs of AAI Test 7. This is the output we would have expected for Test 7 of the AAI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is because the target voltage (10V) exceeds the maximum voltage (5V) of capacitor C21 in the pacemaker so it will not send pulses to the heart.

Test 8: Displaying AAI Bradycardia Therapy at 60bpm with a lower limit atrial refractory period of 150ms while heart is at 30bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	10	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	150

Table 24: the pacemaker and Heartview inputs for Test 8; testing the AAI mode at 60bpm with an ARP of 150ms while heart is at 30bpm.

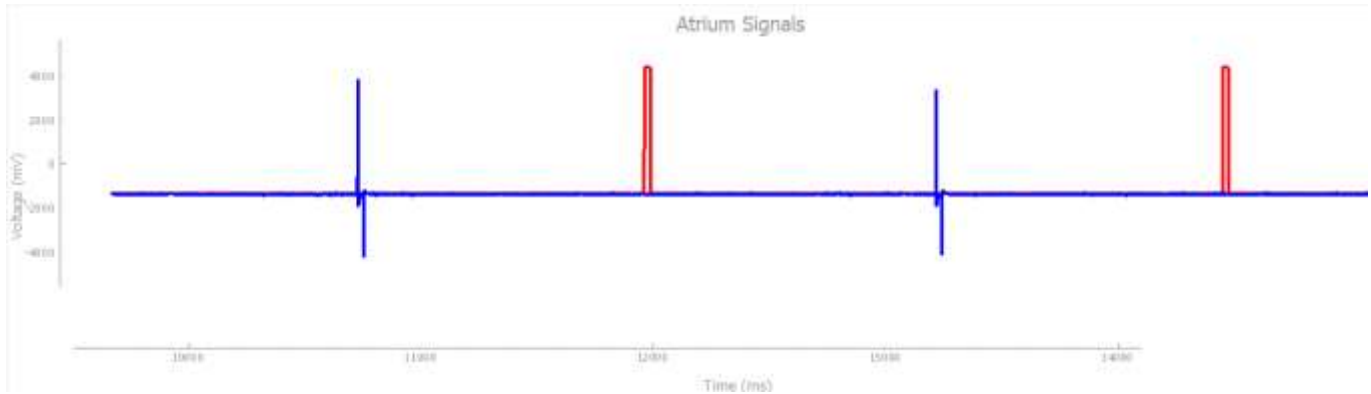


Figure 19: the Heartview output for Test 8; testing the AAI mode at 60bpm with an ARP of 150ms while heart is at 30bpm.

Figure 19 displays the outputs of AAI Test 8. AAI Test 8 is essentially the same as AAI Test 1, so the same result is expected. The exception is that it has a refractory period of 150ms which is the lower limit for the atrial refractory period (Table 1). This is a boundary test case to ensure that the pacemaker will still provide a pulse when the refractory period is at its lowest.

Test 9: Displaying AAI Bradycardia Therapy at 60bpm with an upper limit atrial refractory period of 500ms while heart is at 30bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	ON
Target BPM (bpm)	60	Natural Ventricle	OFF
Target Output Voltage (V)	10	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	500

Table 25: the pacemaker and Heartview inputs for Test 9; testing the AAI mode at 60bpm with an ARP of 500ms while heart is at 30bpm.

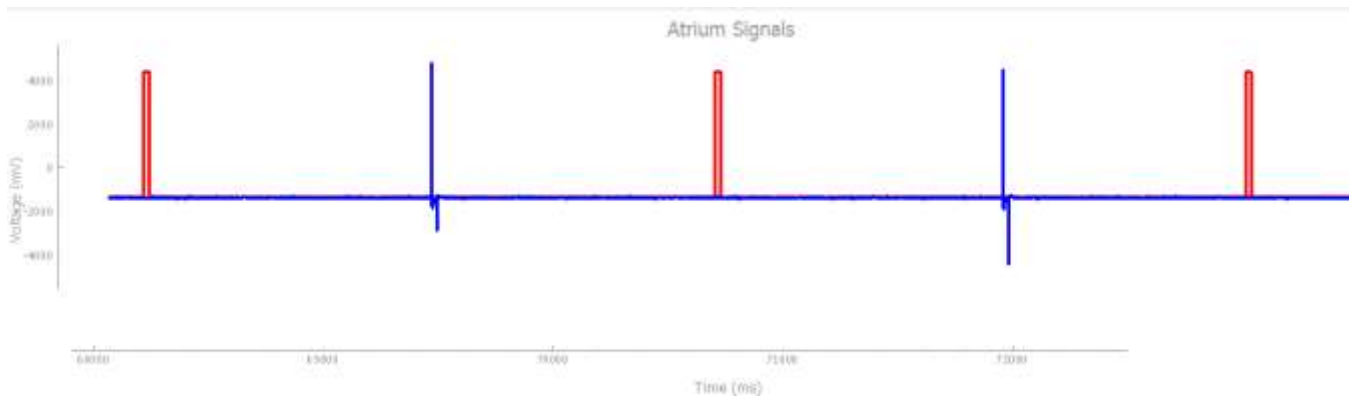


Figure 20: the Heartview output for Test 9; testing the AAI mode at 60bpm with an ARP of 500ms while heart is at 30bpm.

Figure 20 displays the outputs of AAI Test 9. AAI Test 9 is essentially the same as AAI Test 1 and 8, so the same result is expected. The exception is that it has a refractory period of 500ms which is the upper limit for the atrial refractory period (Table 1). This is a boundary test case to ensure that the pacemaker will still provide a pulse when the refractory period is at its highest.

VVI Testing

- Refractory period was set to 320ms for VVI testing because that is the value which was specified for Ventricular Refractory Period in Table 1.

Test 1: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 30bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 26: the pacemaker and Heartview inputs for Test 1; testing the VVI mode at 60bpm while heart is at 30bpm.

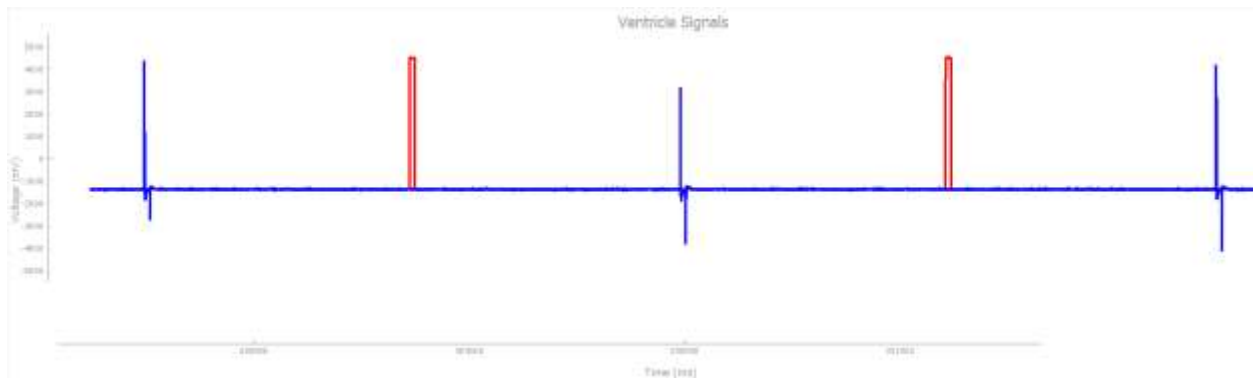


Figure 21: the Heartview output for Test 1; testing the VVI mode at 60bpm while heart is at 30bpm.

Figure 21 displays the outputs of VVI Test 1. This is the output we would have expected for Test 1 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 30bpm and blue pulses (pacemaker) paced at 60bpm. The target bpm is 60bpm however the heart is only pacing at 30bpm. Therefore, the pacemaker needs to pace the ventricles in between natural heart beats to compensate for the slow natural heart rate.

Test 2: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 60bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	60
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 27: the pacemaker and Heartview inputs for Test 2; testing the VVI mode at 60bpm while heart is at 60bpm.

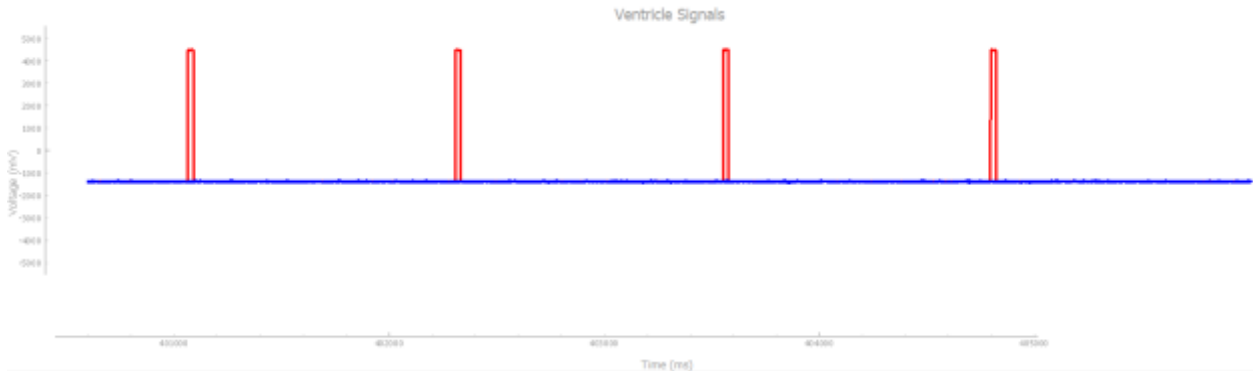


Figure 22: the Heartview output for Test 2; testing the VVI mode at 60bpm while heart is at 60bpm.

Figure 22 displays the outputs of VVI Test 2. This is the output we would have expected for Test 2 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). The blue pulses do not appear because the heart is naturally beating at the target bpm so there is no reason for the pacemaker to intervene.

Test 3: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 59bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	59
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 28: the pacemaker and Heartview inputs for Test 3; testing the VVI mode at 60bpm while heart is at 59bpm.

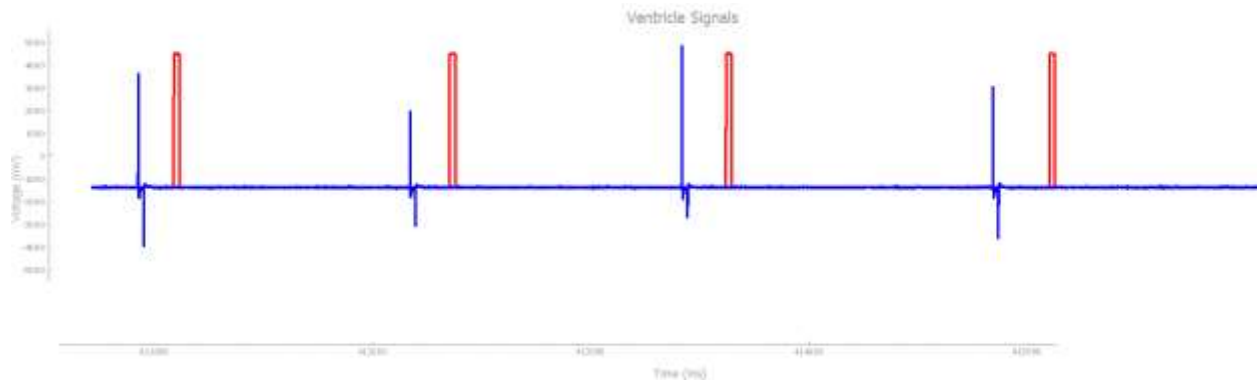


Figure 23: the Heartview output for Test 3; testing the VVI mode at 60bpm while heart is at 59bpm.

Figure 23 displays the outputs of VVI Test 3. This is the output we would have expected for Test 3 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and blue pulses (pacemaker). This is a boundary test case to ensure that the pacemaker is providing a pulse to the heart even when it is just shy of the target bpm (60bpm).

Test 4: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 61bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	61
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 29: the pacemaker and Heartview inputs for Test 4; testing the VVI mode at 60bpm while heart is at 61bpm.

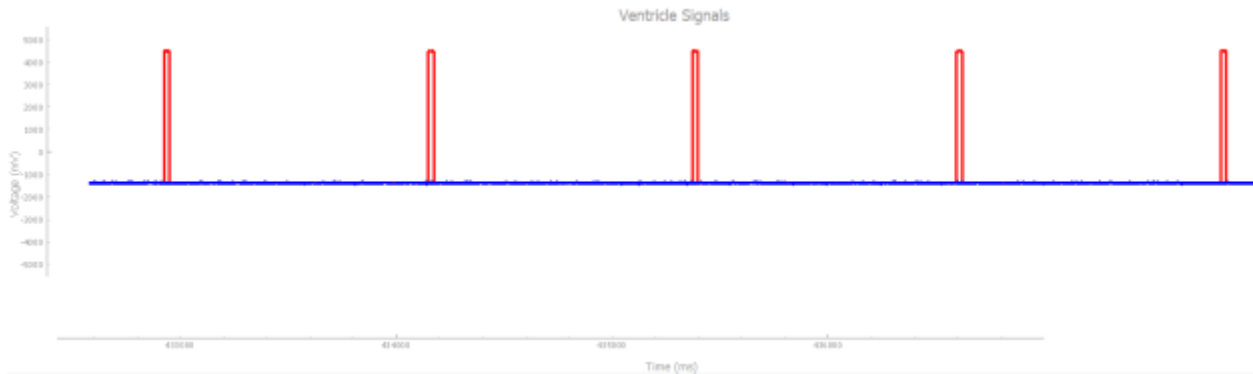


Figure 24: the Heartview output for Test 4; testing the VVI mode at 60bpm while heart is at 61bpm.

Figure 24 displays the outputs of VVI Test 4. This is the output we would have expected for Test 4 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is a boundary test case to ensure that the pacemaker is not providing a pulse to the heart even when it is just above the target bpm (60bpm).

Test 5: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 120bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	120
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 30: the pacemaker and Heartview inputs for Test 5; testing the VVI mode at 60bpm while heart is at 120bpm.

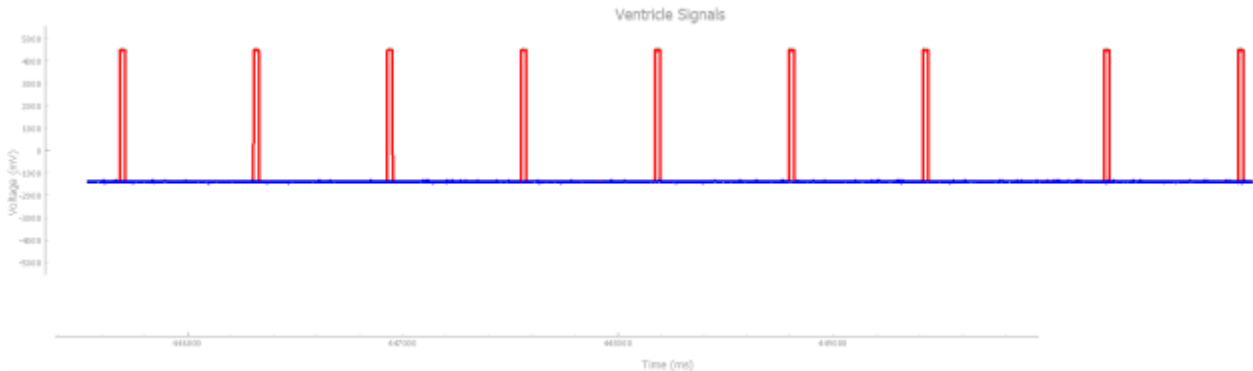


Figure 25: the Heartview output for Test 5; testing the VVI mode at 60bpm while heart is at 120bpm.

Figure 25 displays the outputs of VVI Test 5. This is the output we would have expected for Test 5 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is because the heart is already naturally pacing at above its target bpm.

Test 6: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 30bpm while button SW2 is pressed.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	3	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 31: the pacemaker and Heartview inputs for Test 6; testing the VVI mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.

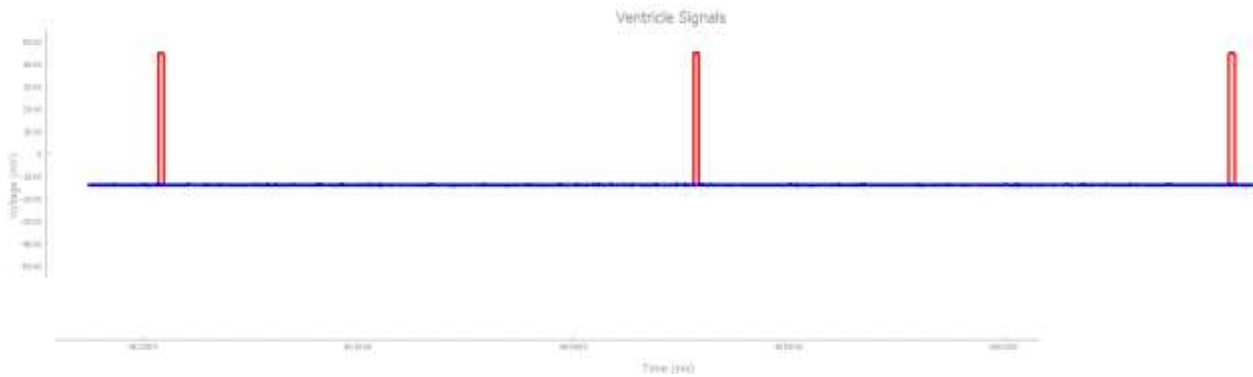


Figure 26: the Heartview output for Test 6; testing the VVI mode at 60bpm while heart is at 30bpm and SW2 button is being pressed.

Figure 26 displays the outputs of VVI Test 6. This is the output we would have expected for Test 6 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is because the pulses from the pacemaker inhibited due to the SW2 button being pressed as written in our Simulink model.

Test 7: Displaying VVI Bradycardia Therapy at 60bpm while heart is at 30bpm and a target voltage of 10V.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	3	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	10	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	320

Table 32: the pacemaker and Heartview inputs for Test 7; testing the VVI mode at 60bpm while heart is at 30bpm and a target voltage of 10V.

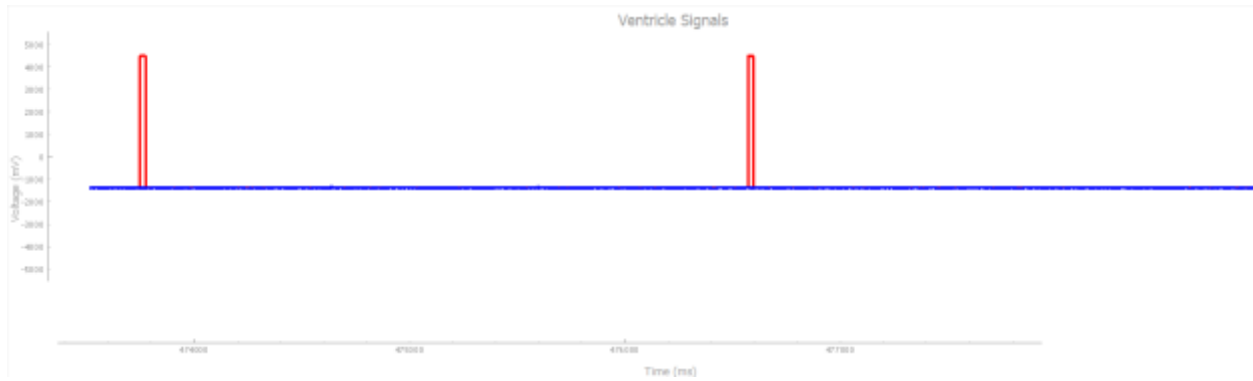


Figure 27: the Heartview output for Test 7; testing the VVI mode at 60bpm while heart is at 30bpm and a target voltage of 10V.

Figure 27 displays the outputs of VVI Test 7. This is the output we would have expected for Test 7 of the VVI mode given the input values into the model and Heartview. The plot displays red pulses (natural heart) paced at 60bpm and with no blue pulses present (pacemaker). This is because the target voltage (10V) exceeds the maximum voltage (5V) of capacitor C21 in the pacemaker so it will not send pulses to the heart.

Test 8: Displaying VVI Bradycardia Therapy at 60bpm with a lower limit atrial refractory period of 150ms while heart is at 30bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	4	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	10	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	150

Table 33: the pacemaker and Heartview inputs for Test 8; testing the VVI mode at 60bpm with an VRP of 150ms while heart is at 30bpm.

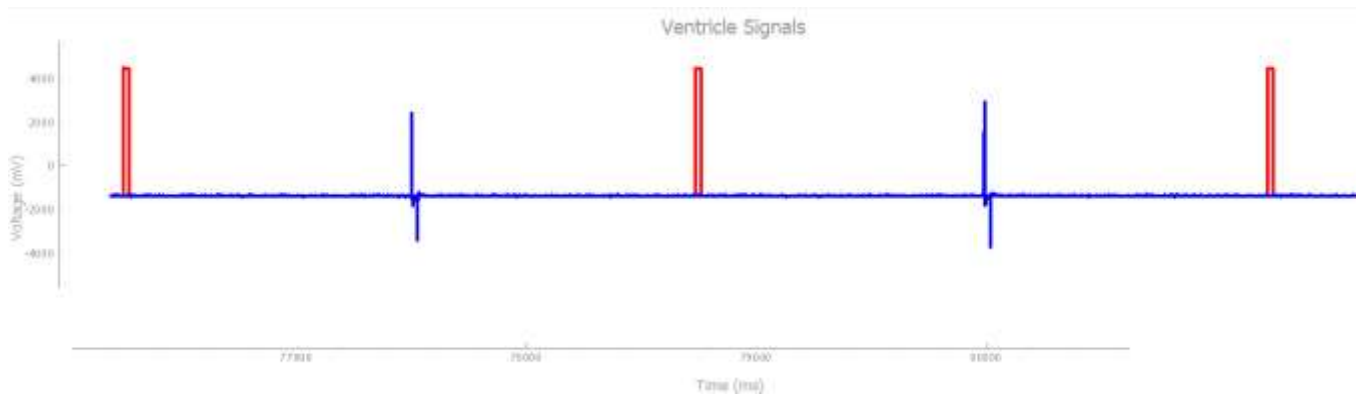


Figure 28: the Heartview output for Test 8; testing the VVI mode at 60bpm with an VRP of 150ms while heart is at 30bpm.

Figure 28 displays the outputs of VVI Test 8. VVI Test 8 is essentially the same as AAI Test 1, so the same result is expected. The exception is that it has a refractory period of 150ms which is the lower limit for the ventricular refractory period (Table 1). This is a boundary test case to ensure that the pacemaker will still provide a pulse when the refractory period is at its lowest.

Test 9: Displaying VVI Bradycardia Therapy at 60bpm with an upper limit atrial refractory period of 500ms while heart is at 30bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	4	Natural Atrium	OFF
Target BPM (bpm)	60	Natural Ventricle	ON
Target Output Voltage (V)	10	Natural Heart Rate (bpm)	30
Pulse Duration (ms)	20	Natural AV Delay	N/A
		Refractory Period (ms)	500

Table 34: the pacemaker and Heartview inputs for Test 9; testing the VVI mode at 60bpm with an ARP of 500ms while heart is at 30bpm.

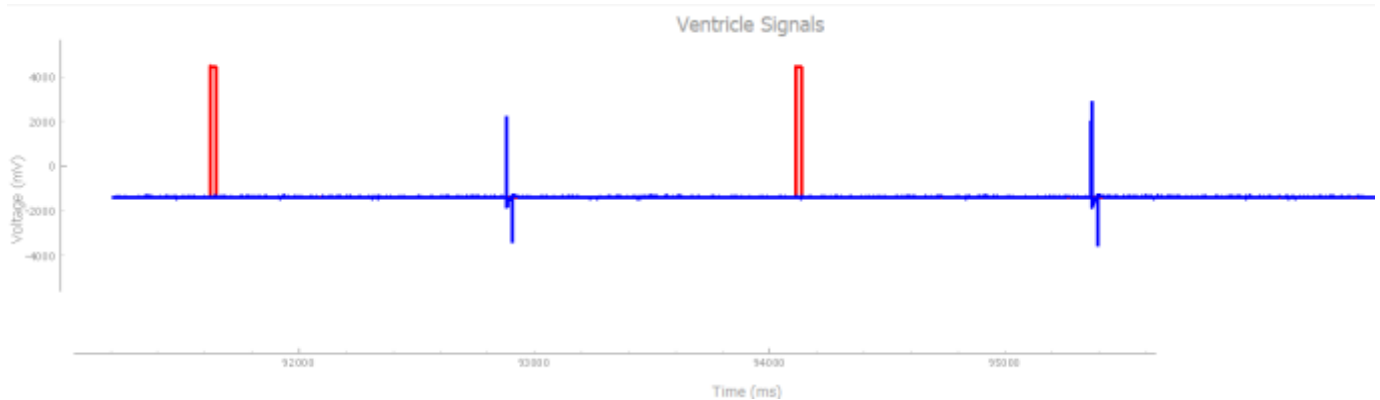


Figure 29: the Heartview output for Test 9; testing the AA1 mode at 60bpm with an ARP of 500ms while heart is at 30bpm.

Figure 29 displays the outputs of VVI Test 9. VVI Test 9 is essentially the same as VVI Test 1 and 8, so the same result is expected. The exception is that it has a refractory period of 500ms which is the upper limit for the ventricular refractory period (Table 1). This is a boundary test case to ensure that the pacemaker will still provide a pulse when the refractory period is at its highest.

AOOR Testing

Test 1: Displaying AOOR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	7	Natural Atrium	OFF
Lower Rate Limit (LRL)	60	Natural Ventricle	OFF
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	250

Table 35: the pacemaker and Heartview inputs for Test 1; testing the AOOR mode with a LRL of 60bpm and URL of 120bpm.

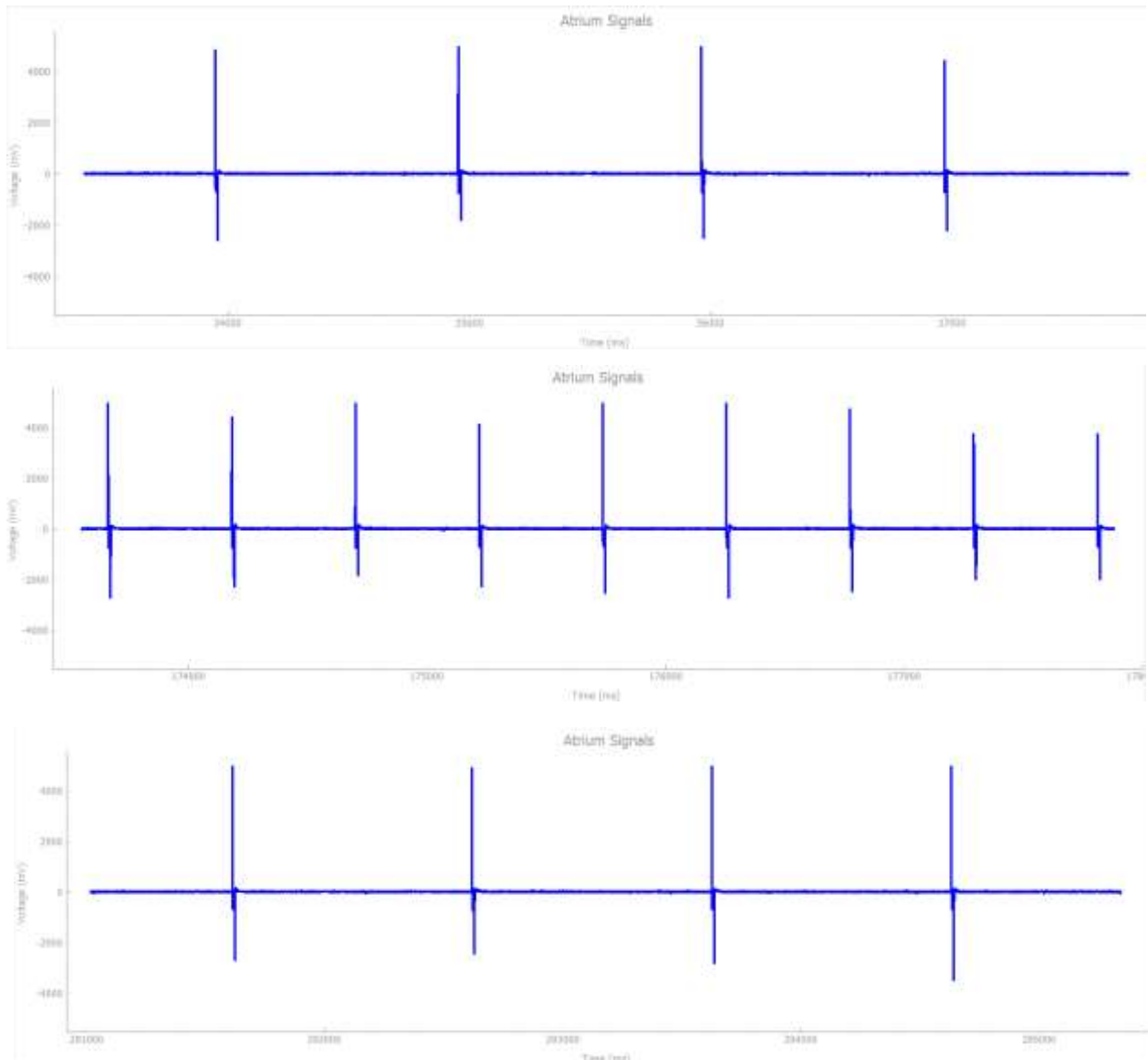


Figure 30: Testing AOOR with a LRL of 60bpm and URL of 120bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

VOOR Testing

Test 1: Displaying VOOR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	5	Natural Atrium	OFF
Lower Rate Limit (LRL)	60	Natural Ventricle	OFF
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	320

Table 36: the pacemaker and Heartview inputs for Test 1; testing the VOOR mode with a LRL of 60bpm and URL of 120bpm..

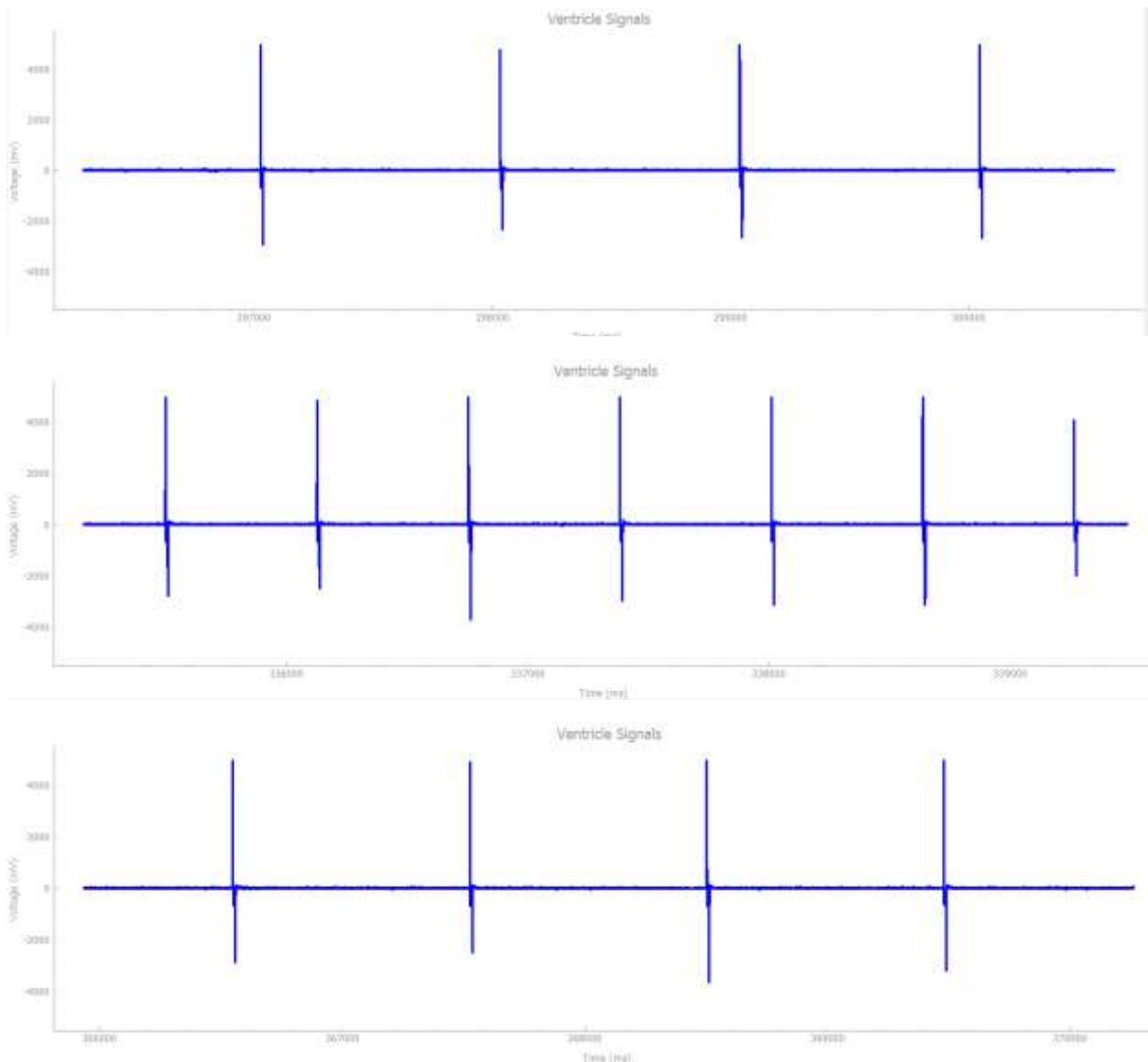


Figure 31: Testing VOOR with a LRL of 60bpm and URL of 120bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

AAIR Testing

Test 1: Displaying AAIR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm while the natural heart is set at 90 bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	8	Natural Atrium	ON
Lower Rate Limit (LRL)	60	Natural Ventricle	OFF
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	250

Table 37: the pacemaker and Heartview inputs for Test 1; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.

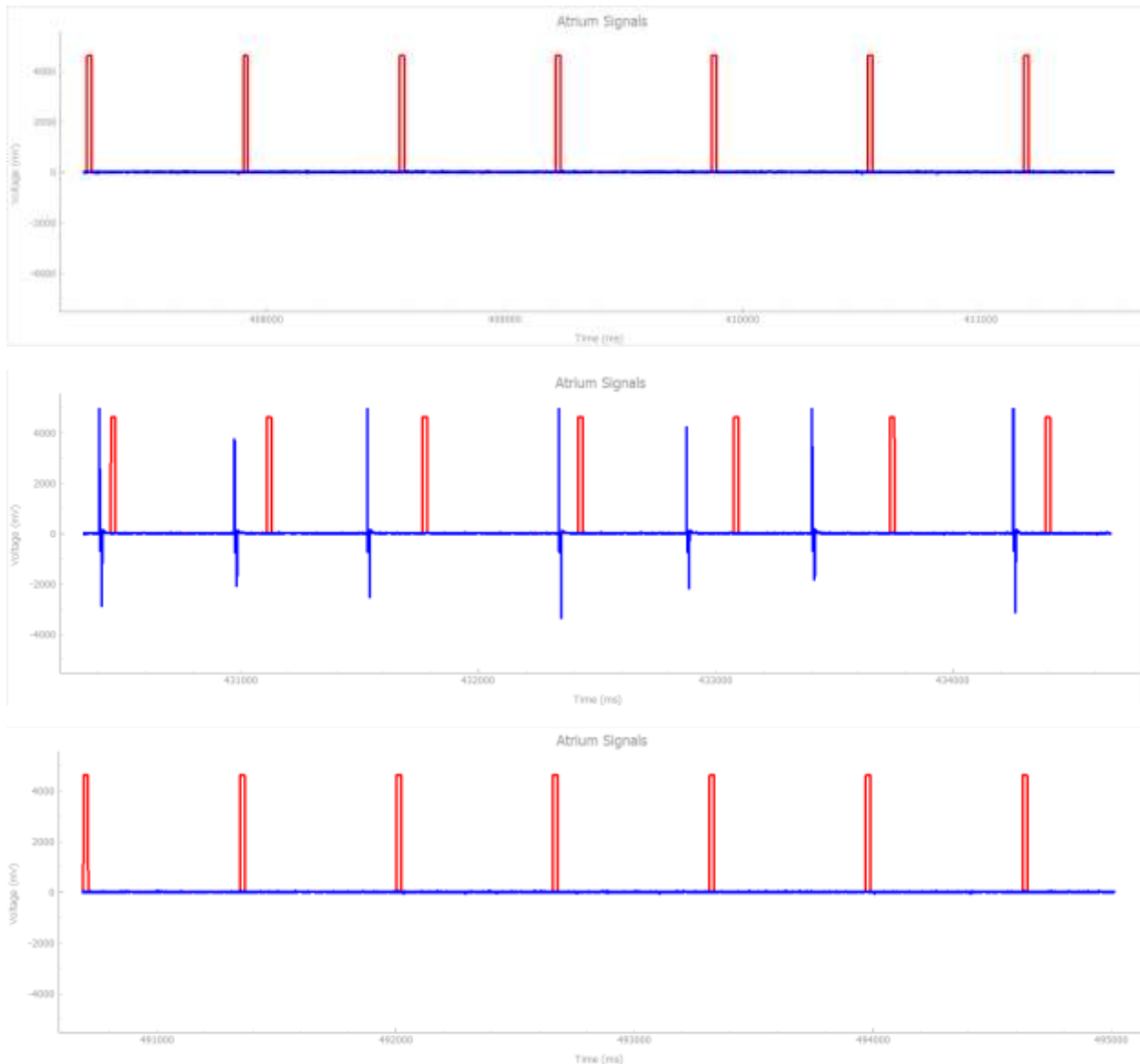


Figure 32: Testing AAIR with a LRL of 60 bpm and URL of 120bpm while the heart is at 90bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

Test 2: Displaying AAIR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm while the natural heart is set at 30 bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	8	Natural Atrium	ON
Lower Rate Limit (LRL)	60	Natural Ventricle	OFF
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	250

Table 38: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.

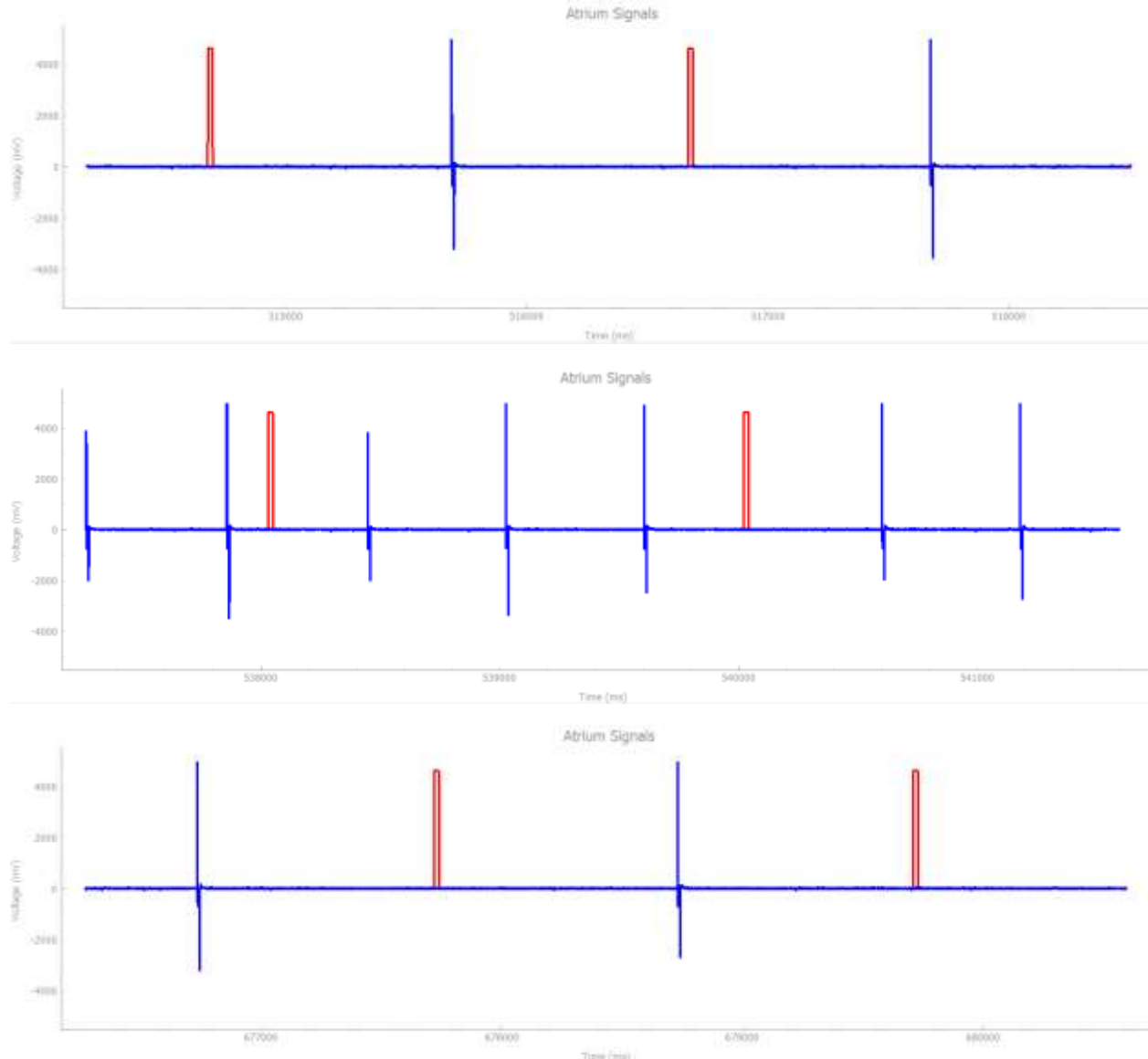


Figure 33: Testing AAIR with a LRL of 60bpm and URL of 120bpm while the heart is at 30bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

Test 3: Displaying AAIR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm while the natural heart is set at 140 bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	8	Natural Atrium	ON
Lower Rate Limit (LRL)	60	Natural Ventricle	OFF
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	250

Table 39: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.

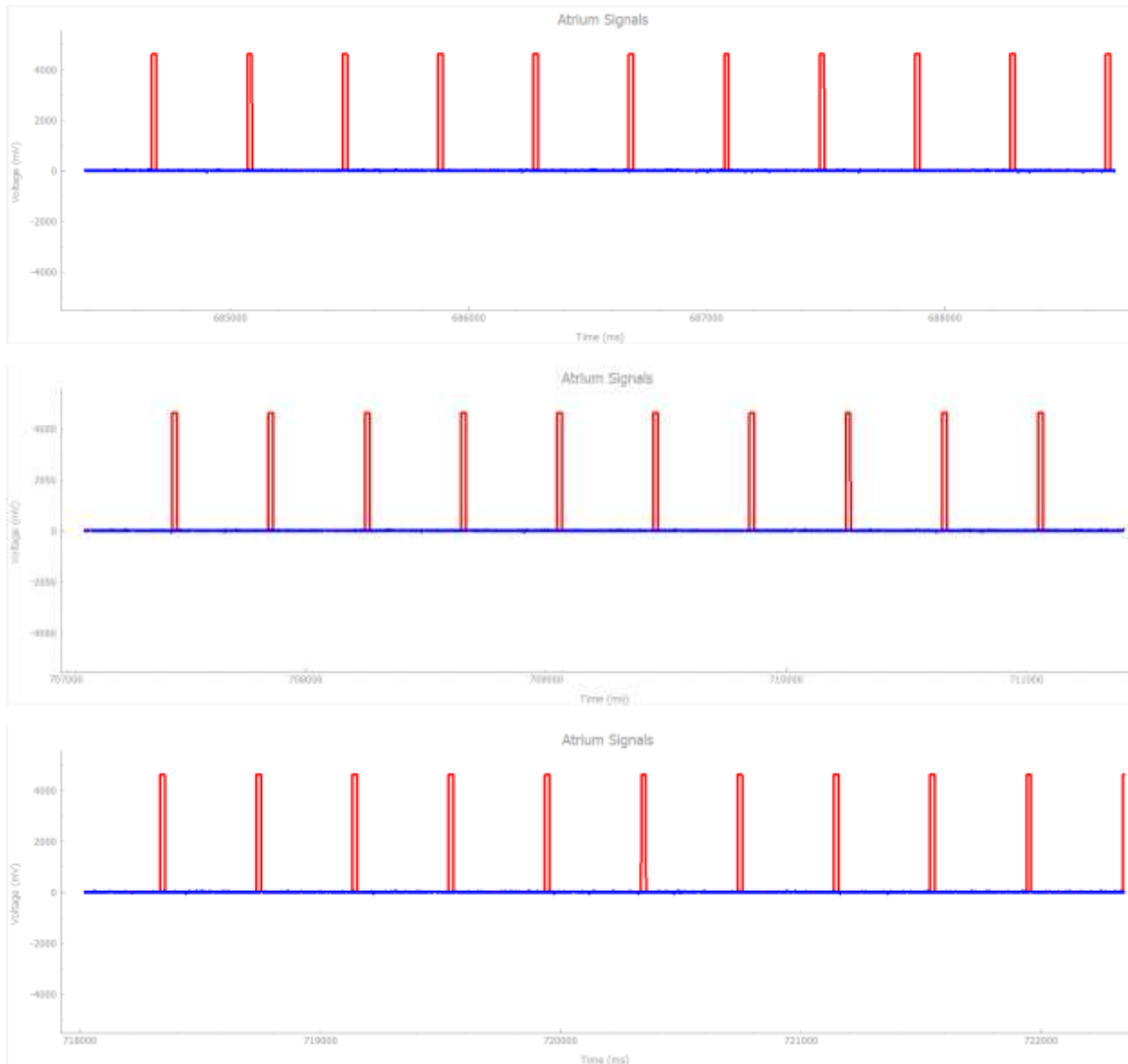


Figure 34: Testing AAIR with a LRL of 60bpm and URL of 120bpm while the heart is at 140bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

VVIR Testing

Test 1: Displaying VVIR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm while the natural heart is set at 90 bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	6	Natural Atrium	OFF
Lower Rate Limit (LRL)	60	Natural Ventricle	ON
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	320

Table 40: Table 37: the pacemaker and Heartview inputs for Test 1; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.

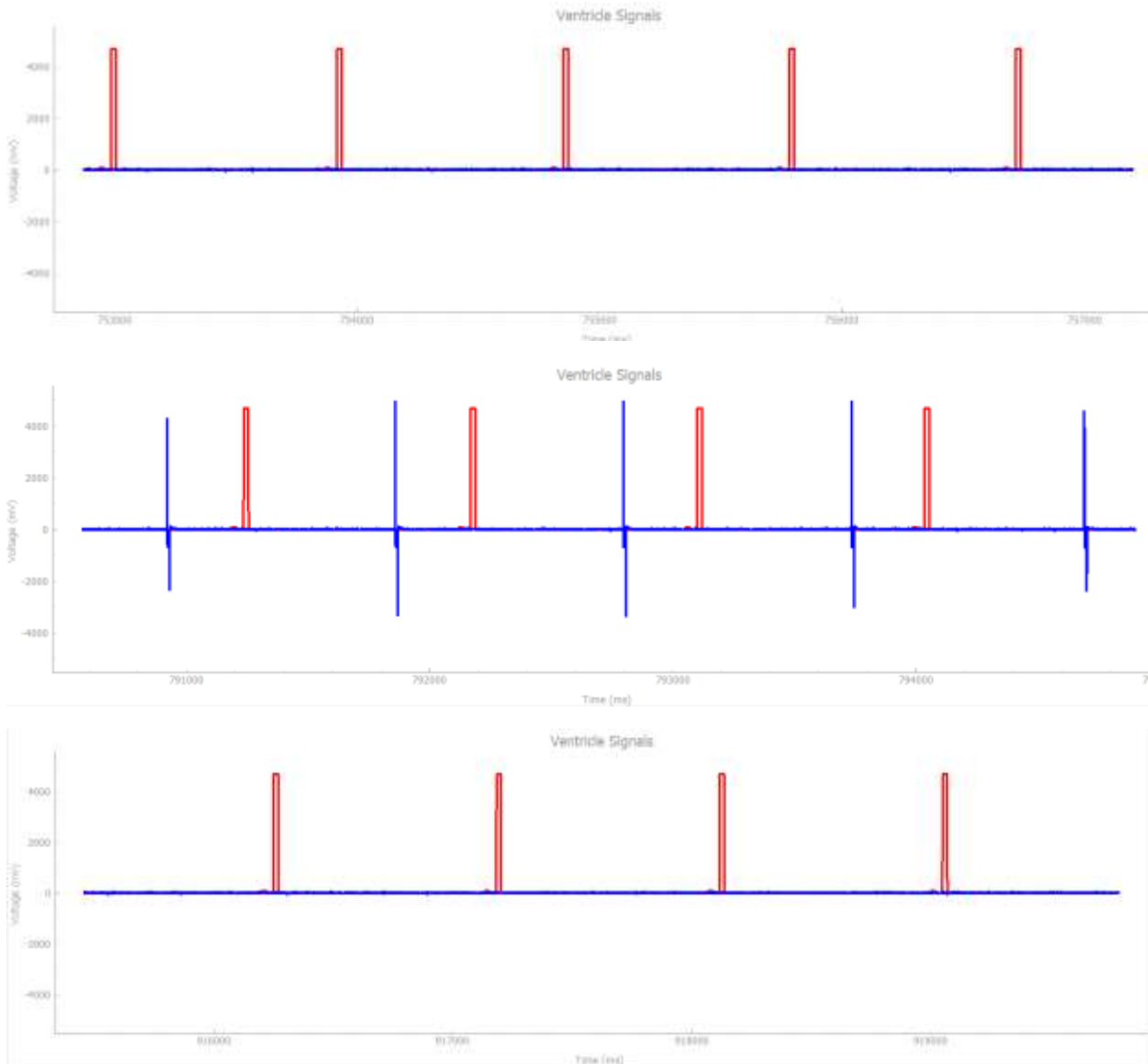


Figure 35: Testing VVIR with a LRL of 60bpm and URL of 120bpm while the heart is at 60bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

Test 2: Displaying VVIR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm while the natural heart is set at 30 bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	6	Natural Atrium	OFF
Lower Rate Limit (LRL)	60	Natural Ventricle	ON
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	320

Table 41: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.

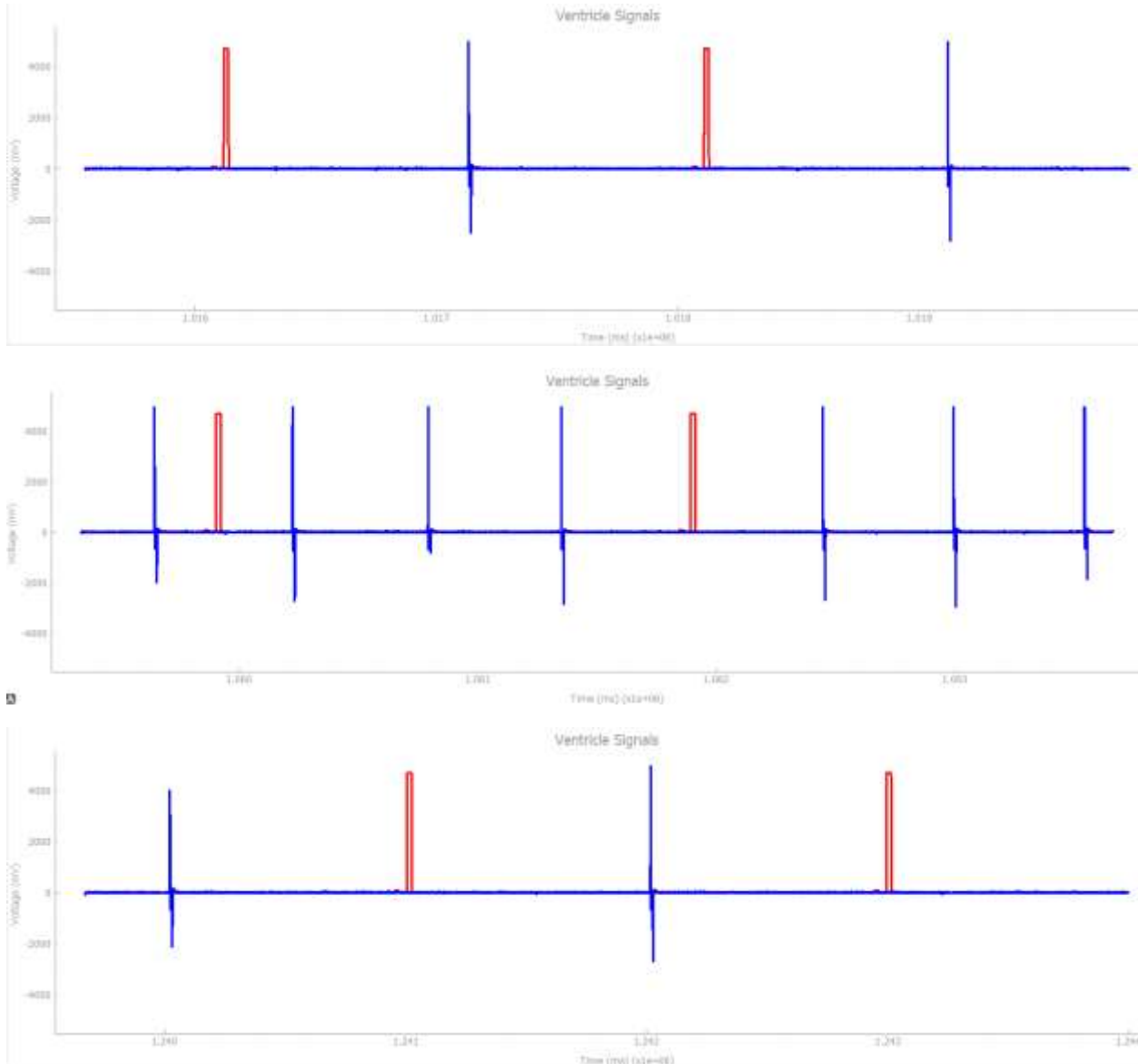


Figure 36: Testing VVIR with a LRL of 60bpm and URL of 120bpm while the heart is at 30bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

Test 3: Displaying VVIR Bradycardia Therapy with a lower rate limit of 60bpm and an upper rate limit of 120bpm while the natural heart is set at 140 bpm.

Pacemaker Inputs	Values	Heartview Inputs	Values
Mode	6	Natural Atrium	OFF
Lower Rate Limit (LRL)	60	Natural Ventricle	ON
Upper Rate Limit (URL)	120	Natural Heart Rate (bpm)	N/A
Reaction Time (s)	15	Natural AV Delay	N/A
Recovery Time (minutes)	2	Refractory Period (ms)	320

Table 42: the pacemaker and Heartview inputs for Test 2; testing the AAIR mode with a LRL of 60bpm and URL of 120bpm.

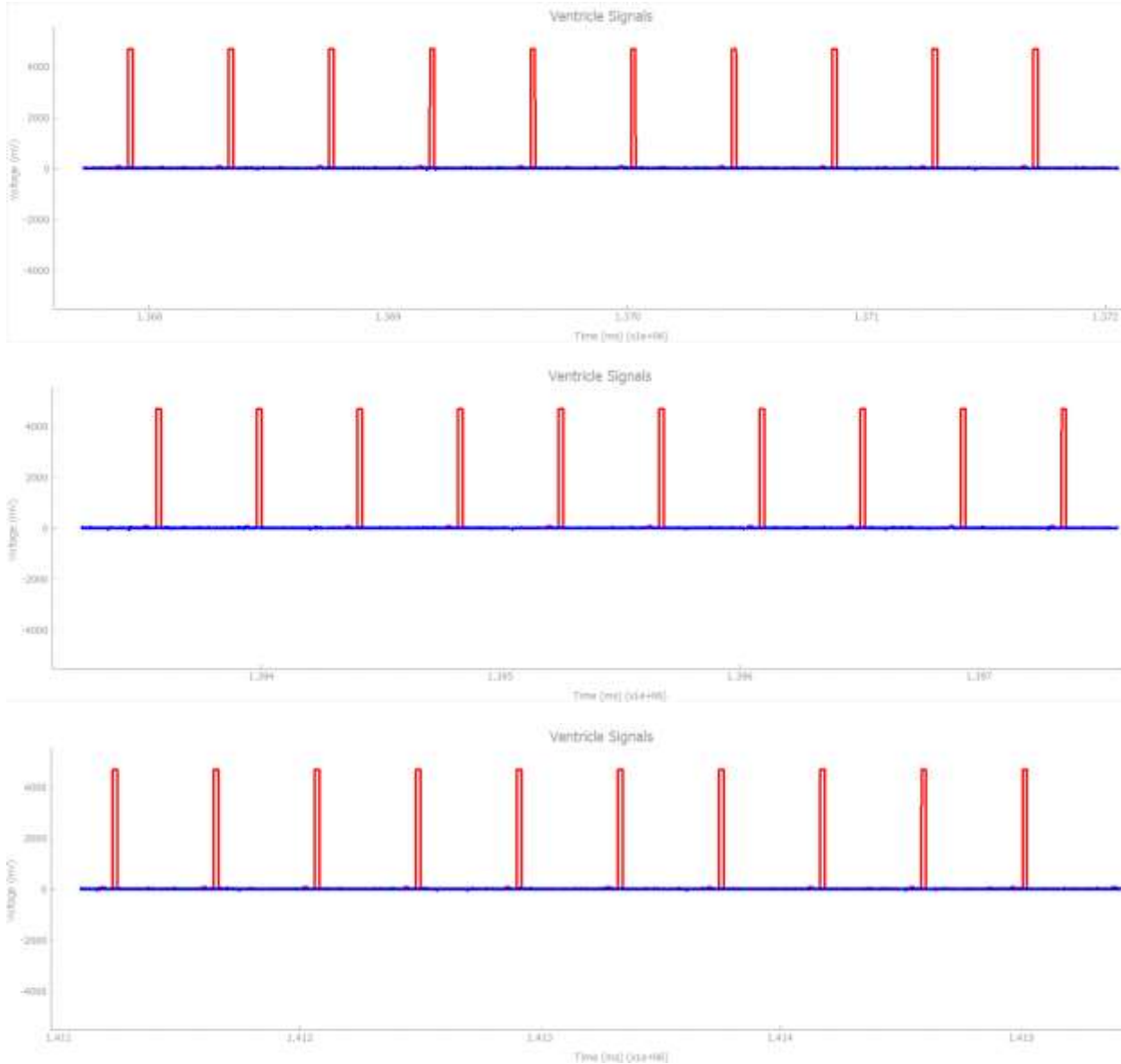


Figure 37: Testing VVIR with a LRL of 60bpm and URL of 120bpm while the heart is at 140bpm. Top: resting state (no activity). Middle: shaking the pacemaker (activity). Bottom: stopped shaking the pacemaker (returned to no activity).

Future Requirement Changes

In the future, the requirements for the pacemaker that are likely to change are:

- The addition of new modes including DDD, DDDR.
- The pacemaker must be able to pace and sense the atria and ventricles at the same time while adjusting the rate.
- The pacemaker should inhibit only ventricular pacing when holding down the pushbutton.

Future Design Changes

Design changes will have to be made to account for the likely requirement changes stated above:

One of the issues that arose throughout the testing of the pacemaker was the processing speed of our program. One of the contributing factors is the hardware and software that we are using. Simulink generates C code based upon the state flow, so we have limited influence over the actual code. If we had access to the full code, we could have used uint8 datatypes for different variables instead of uint16 which would speed up the communication between the heart and the pacemaker and allow for more accurate sensing.

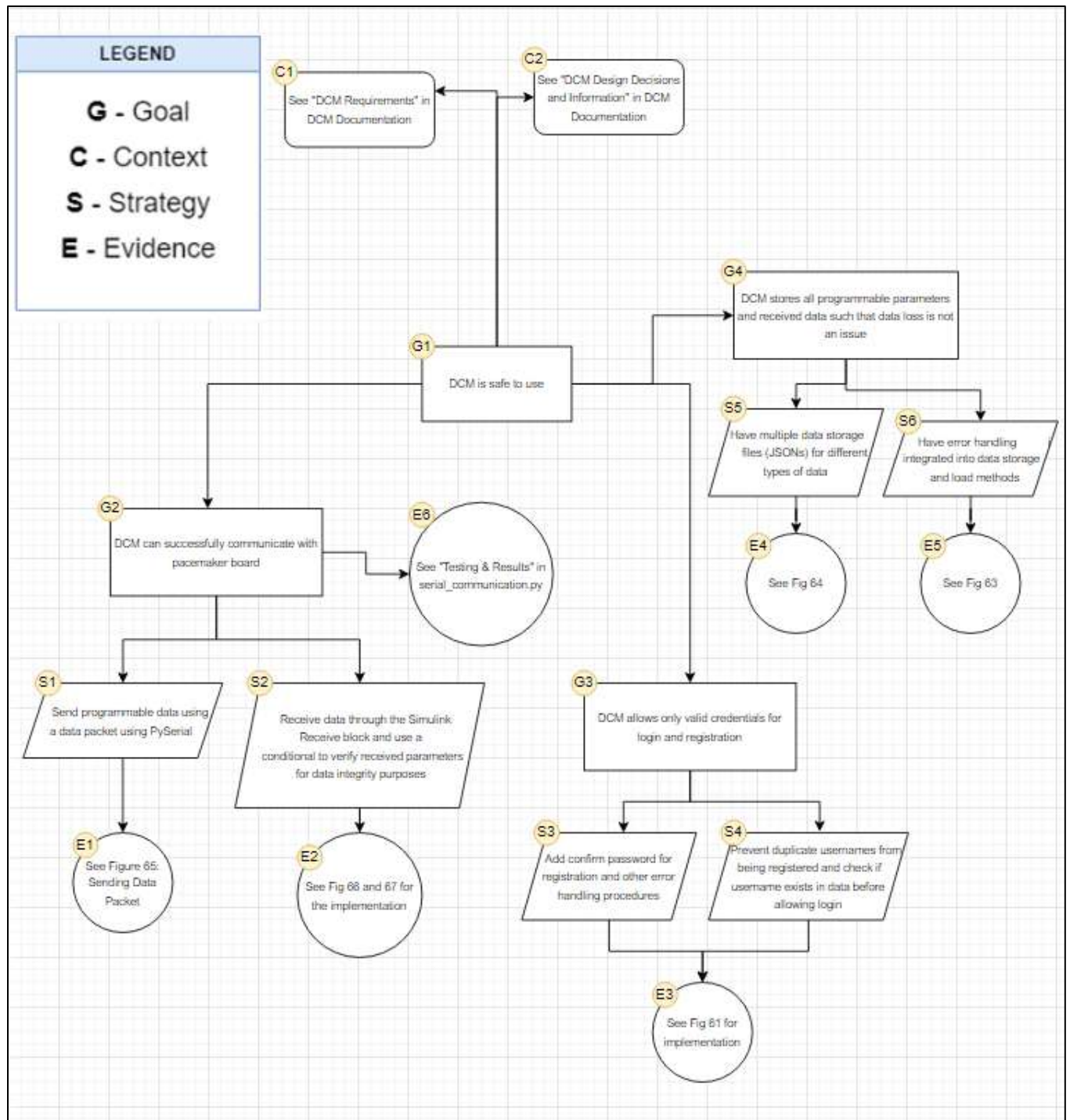
DCM Design Documentation

DCM Requirements

Requirement	Description
1	Develop an interface that includes a welcome screen, including the ability to register a new user (name and password), and to login as an existing user. A maximum of 10 users should be allowed to be stored locally.
2	The user interface shall be capable of utilizing and managing windows for display of text and graphics.
3	The user interface shall be capable of processing user positioning and input buttons
4	The user interface shall be capable of displaying all programmable parameters for review and modification.
5	The user interface shall be capable of visually indicating when the DCM and the device are communicating.
6	The user interface shall be capable of visually indicating when a different PACEMAKER device is approached than was previously interrogated.
7	Develop interfaces to present all the pacing modes.
8	Make provision for storing programmable parameter data for checking inputs. The system will be able to set, store, transmit programmable parameter data, and verify it is stored correctly on the Pacemaker device
9	Develop and document appropriate data structures for electrogram data required in future assignments. The system will be able to display electrogram data when the user chooses to do so (for either ventricle, atrium, or both).
10	The DCM must receive the electrogram data from the Pacemaker over the serial communication link to display it.
11	Document how programmable parameters originate at the DCM and are implemented in the device. Show how you can ensure the parameters stored in the Pacemaker are what the doctor input on the DCM. Also justify your choice of the data types used to represent parameters data.

Table 4143: DCM Requirements

DCM Safety Assurance Case Diagram



DCM Design Decisions and Information

The goal of this section is to explain the general process we followed to develop the application for each stage, and the general challenges we faced. This section will serve as a justification of our design choices. Detailed information on the tests we performed, and their results are included with the breakdown of the modules in the next section.

Login & Registration

Before commencing the planning phase, our team reviewed the requirements outlined in the 'Assignment1' document on Avenue to Learn. This document served as a reference, providing our team with a general understanding of what the pacemaker should include in the first phase of development. For the programming language, we decided to use the Tkinter module in Python for the development of our DCM. Web development was the alternative, particularly using HTML, CSS, and Javascript. However, our team decided to use Tkinter for later benefits such as the libraries it provides for serial communication needed for Assignment 2.

The next phase in our project involved comprehensive planning of the product's design. To do this, our team utilized FIGMA to create mock GUI templates, enabling us to visualize the final product. Each team member contributed to this design by conducting individual research and inspecting various websites featuring user sign-in interfaces, providing valuable inspiration for shaping our product's design. Upon completing the FIGMA model, our team transitioned to the process of understanding the Tkinter module and initiating the development of our DCM. As expected, the initial stages presented a learning curve, especially for those team members who had limited prior experience with Tkinter. However, this challenge was effectively addressed, as all team members dedicated time to practice and enhance their Tkinter proficiency independently.

During the process, one of the first roadblocks that the team encountered was the geometry handling with the widgets. A widget is basically Tkinter's name for elements in the GUI (for example, buttons, labels, text boxes/entries). Within Tkinter, there are three methods of geometry management which are `grid()`, `pack()`, and `place()`. During the first stage of the process, the team had chosen to use `grid()` as they assumed that the `grid()` method would allow them to resize the window size while all the widgets would stay in place relative to the boundary of the window. However, later during design, the team came to the realization that Tkinter's `grid()` method is a bit different from other languages where the `grid()` method would not adjust according to the window size meaning certain widgets would be off-screen whenever the window is small enough.

We saw this as an immediate issue and thought of doing further research into what we may use for our widget geometry management. During our research, we found that using `pack()` and `place` conjointly would perfectly work with what we were looking for in terms of the design as well as the functionality of the application. The team had found that the `pack()` method would ultimately aid us in controlling the commands of the buttons and the `place` method would help us where to physically place the widgets within the application. Thus, our plan was to create frames within the application which would be packed in order to display or not display certain screens within the application. Using the `place` method, we would insert the widgets as to where they should be in those individual frames. This allowed us to unpack or pack a certain frame whenever needed using `frame.pack()` and `frame.pack_forget` method which made our task of commanding the buttons much easier. For demonstration, within our code, we have separate frames for sign-in and registration, where if the user were to click on the register button, it would unpack the login frame and pack the register frame so the user is displayed with the registration screen. This method allowed us to easily have control of what the user sees according to which button

they press. This is the general method we used for routing. We made a route() function in our main module which forgets all frames except the target frame allowing for navigation.

Below is an example of how our DCM GUI design turned out versus what we planned in FIGMA:



Figure 38: FIGMA Model of the sign in interface



Figure 39: Implementation of Login Page Based on FIGMA Model



Figure 40: Implementation of Registration Page Based on FIGMA Model

After dealing with the challenges of geometry and familiarity with Tkinter, we had to deal with the challenges of storing user data that is entered. For the functionality of this process, we decided to use JSON (JavaScript Object Notation) files as our storage method. They function almost the same as text files but with a dictionary (key, value) pair method of storage.

Now that we had decided on the method of storage, we decided to create our second module (first one was main.py) called user_manager.py. This module would handle the login and registration process as well as accessing the data from our user_data.json (stores username/password of users). Error handling was implemented in this module for requirements such as only allowing a maximum of 10 users to be stored. We decided not to let the existing users be deleted from within the GUI for security reasons. After running test cases on the login/registration process, we had successfully implemented the login/registration process of the program. One of the major issues we faced when testing was actually a very simple problem in disguise:

Initially, register wasn't working because we forgot to add register in the calls and that was invalid because it was using the entries from our login screen which were empty so it would always return 'Username and Password are required' as an error because it was looking at the empty entries from the login frame.

```
def register(self):
    username = self.register_username_entry.get()
    password = self.register_password_entry.get()
    confirm = self.confirm_password_entry.get()
    user_manager.register_user(username,password,confirm)
```

Figure 41: Fix of Using Register Entry instead of Login Entry

Pacemaker Selection

The next step after logging in was allowing the user to select/connect a pacemaker. Before we continued to design the pacemaker interface, we decided to modularize our code by adding a new module called **pacemaker_interface** (this would include the screen of the page after logging in as well as pacemaker log data (which pacemaker was last connected and the saved parameters associated with it)).

After modularizing our code into one main module and two submodules, we had yet another issue where our functions that had multiple arguments would have issues with routing. To resolve this, we had to add the following:

```
command=lambda: self.main.route(self.main.login_frame);
```

Figure 42: Fix for Routing Issue for Functions with Multiple Arguments

The 'command = lambda' part delays the execution of the function until the button is clicked.

```
ImportError: cannot import name 'ModeSel' from partially initialized module 'mode_sel' (most likely due to a circular import) (c:\Users\wobur\OneDrive\Desktop\Pacemaker\PacemakerProject\DCM\mode_sel.py)
```

Figure 43: Circular Import Error

Upon trying to use this command we encountered another error when trying to import the main module into the submodules:

Circular imports occur when two or more modules import each other directly or indirectly, creating a loop in the import structure. To fix this, we had to create an instance of the main class and send it in to the constructor of the submodule as an argument as follows:

```
if __name__ == "__main__":  
    app = SimpleLoginApp()  
    user_manager = UserManager("DCM/DataStorage/user_data.json", app)  
    pacemaker_interface = PacemakerInterface(app.pacemaker_sel, app)
```

Figure 44: Sending an instance of the main module into the constructor of submodule

```
class PacemakerInterface:  
    def __init__(self, root, main_app):  
        self.root = root  
        self.main = main_app
```

Figure 45: Initializing submodules with the main module as an argument

After fixing the issue with accessing the main module within the submodules, we proceeded to develop the pacemaker selection screen. The main requirements were requirement 4 & 7 of Section 3.2.2 in the PACEMAKER document. To deal with requirement 4, which was about indicating when the DCM and Pacemaker were communicating, we made a placeholder label that displays the boolean of whether communication was established.



Figure 46: GUI for Pacemaker Selection Screen

For requirement 7, which was about indicating when a new pacemaker was connected, we decided to display the previous pacemaker and use that in a conditional to compare it to the pacemaker that is currently being connected. This would display a pop-up message warning the user that the pacemaker being connected is different from the one previously connected.

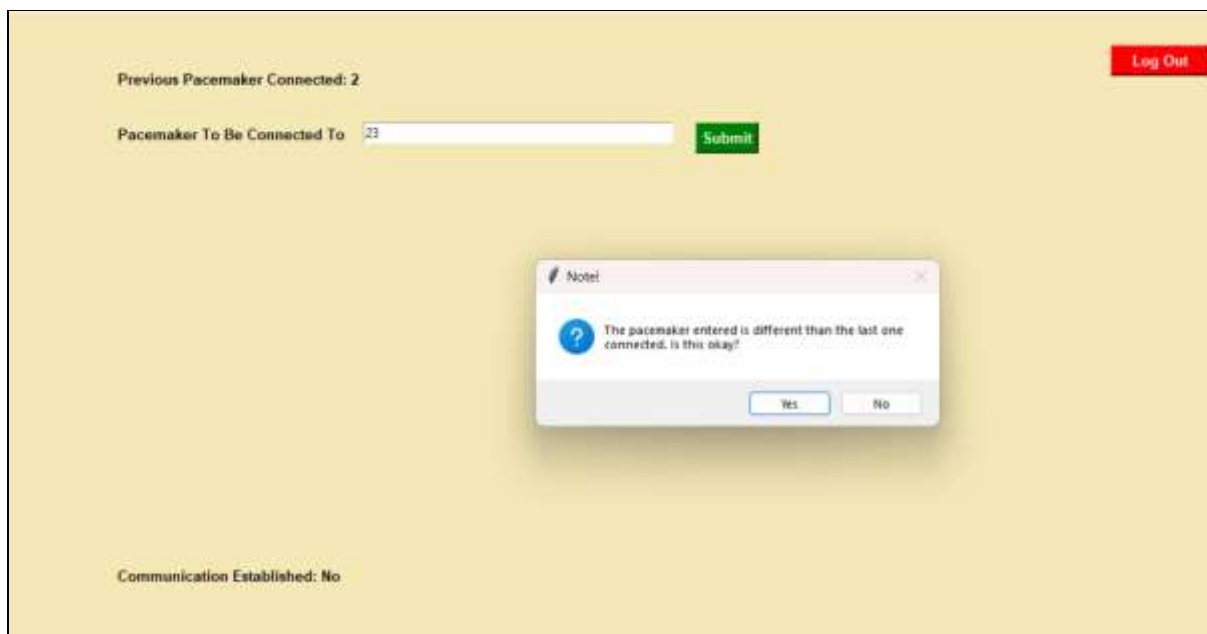


Figure 47: Warning Popup for Different Pacemaker Connection

One of the roadblocks that we had faced at first was an issue with the pacemaker submission where it would accept whitespaces or even nothing within the entry box as a pacemaker submission. This was an issue as we did not want the users to submit these types of entries into the submission as they are not practical names used within the medical industry. To resolve this problem, we had included a separate if statement on to the submit function within the pacemaker_sel.py where it would check whether the entry is either a whitespace or no entry at all before continuing onto the next portion of the submit function where it stores the entered value.

```
def submit(self, entry):  
    if entry.isspace():  
        return messagebox.showerror("Error", "Please enter your Pacemaker")  
    elif entry == "":  
        return messagebox.showerror("Error", "Please enter your Pacemaker")  
    elif self.prev_pacemaker != entry:
```

Figure 48: The solution for whitespace submissions

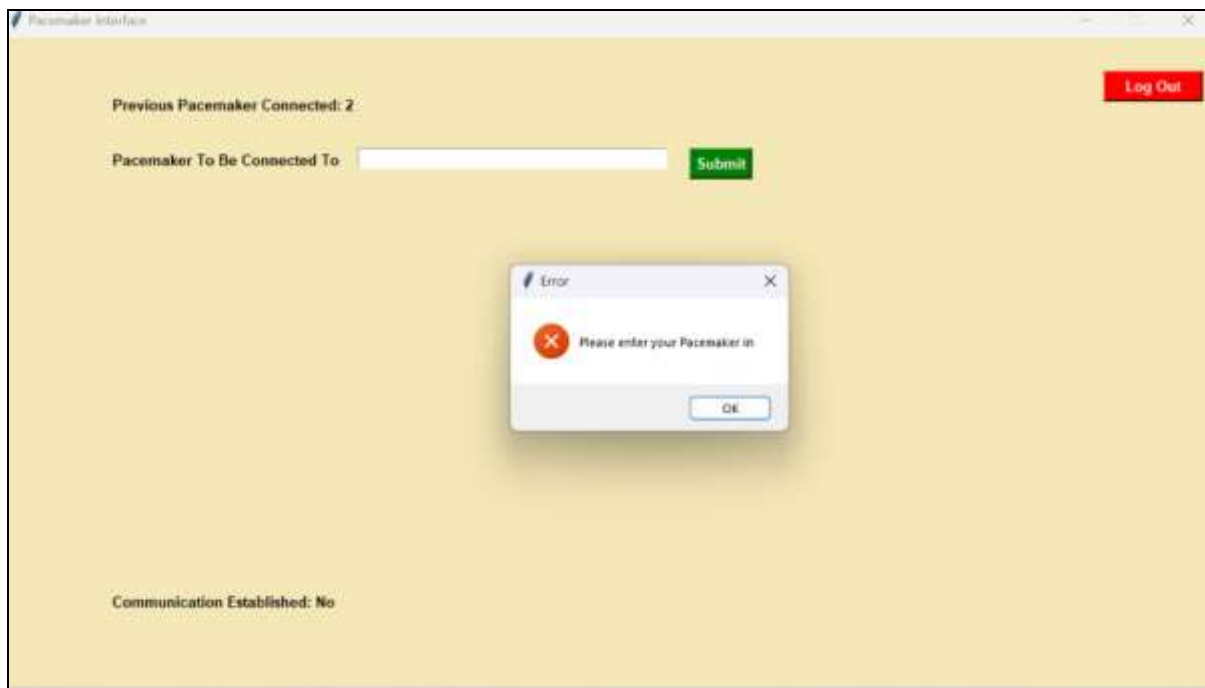


Figure 49: Error Message for Blank Pacemaker Input

Another roadblock we faced in this phase was to decide how we wanted to save the previous pacemaker. For this purpose, we created a new JSON file called previous_pacemaker.json that stores the previous pacemaker connected. Our previous pacemaker label would simply read the value from the JSON file and display it. With this change, we had both the login/registration and pacemaker selection functionalities completed.

Mode Selection

This next module we created called `mode_sel.py` for the purpose of mode selection and to enter parameters was one where we faced the most errors and spent the most time debugging. We started by creating an instance of the class in the main module:

```
mode_selection = ModeSel(app.mode_sel, app)
```

Figure 50: Instance of `mode_sel.py`

For the base GUI of this screen, we decided to have a dropdown menu that lets the user select which mode they want and then upon pressing a button that says 'Next', the corresponding parameters are rendered to let the user enter values for them:



Figure 51: Demonstration of AOO Parameter Rendering

In this fashion, we got the next button to render the relevant parameters depending on the mode for all modes. We tried using sliders instead of entries. This means we would have to manually make increment functions to handle valid values that the slider can go to. After doing this, we got stuck because the sliders would get stuck and start glitching for values smaller than 0.8. We could not fix this issue as this was the limitation of the widget in the tk module (it has the first version of the Tkinter widgets hence the old appearance).



Figure 52: Slider Getting Stuck on Small Value of 0.2

Therefore, we tried using the newer libraries for Tkinter like ttk and ttkbootstrap however neither worked for us. Ttk failed because it did not have a resolution parameter which our increment functions relied on and ttkbootstrap failed because it was not compatible with the other existing widgets which were from the tk library as mentioned earlier. At this point, we decided to go with using entries and just make validation functions for each parameter. To deal with the nominal values of each parameter, we used placeholders as shown in Figure 14 above.

For error handling, we decided to warn the user about invalid inputs through error labels which would render themselves if the user focused out of the entry box. This might seem like a bad approach at first since the user could directly press submit without focusing out. However, we decided to implement message boxes to handle that issue.

 A screenshot of a GUI with a yellow background. It contains four input fields with error messages:

Parameter	Value	Error Message
Lower Rate Limit	131	Lower Rate Limit should be a multiple of 5 between 90ppm and 175ppm.
Upper Rate Limit	213	Upper Rate Limit should be between 50ppm and 175ppm.
Atrial Amplitude	3.523	Amplitude should be a multiple of 0.5 if between 3.5V and 7.0V.
Atrial Pulse Width	0.4232	Pulse Width should be a multiple of 0.1 if between 0.1ms and 1.9ms

 At the bottom, there is a 'Submit Parameters' button.

Figure 53: Error Label Implementation

During the process of creating error labels for the parameters corresponding to each mode, we ran into an issue corresponding to the arithmetic handling within Python. During the process of creating the **validate_amplitude** and **validate_pulse_width** functions within the `model_sel.py` module, we realized that certain values which were within the range of the valid values were being shown as invalid values on our interface. In our function, we use the modulus operator in order to make sure that the values that the user input are incrementing by a certain amount which in this case was 0.1 within the range 0.5V - 3.2V. However, after a numeral amount of testing within Python, we realized that there is an arithmetic rounding occurring with the modulus function which had constantly given us wrong results. This error only occurs when the modulus is of division by 0.1 or a very small number. We tested this for other numbers such as 0.5 and we weren't able to find any error with the modulus operator with this test. Therefore, we steered away from the modulus approach and implemented an iterative approach shown below.

```
def validate_amplitude(self, value):
    try:
        value = float(value)
        if value==0:
            return "Valid"
        elif 0.5<=value<=3.2:
            list = []
            temp = 0.5
            while temp <= 3.3:
                list.append(round(temp,1))
                temp+=0.1
            if value in list:
                return "Valid"
            else:
                return "Amplitude should be a multiple of 0.1 if between 0.5V and 3.2V"
        elif 3.5 <= value <= 7.0:
            if value % 0.5 == 0:
                return "Valid"
            else:
                return "Amplitude should be a multiple of 0.5 if between 3.5V and 7.0V"
        else:
            return "Amplitude should be 0V or between 0.5V - 3.2V or 3.5V - 7.0V"
    except ValueError:
        return "Amplitude should be a valid value."
```

Figure 54: Iterative Solution of Error Checking

After we had created our error labels, we had realized that whenever the user would click the submit parameters button, for invalid values, the application wouldn't react in any way. Thus we made a design decision to create error pop ups which would tell the user what is wrong with the values.

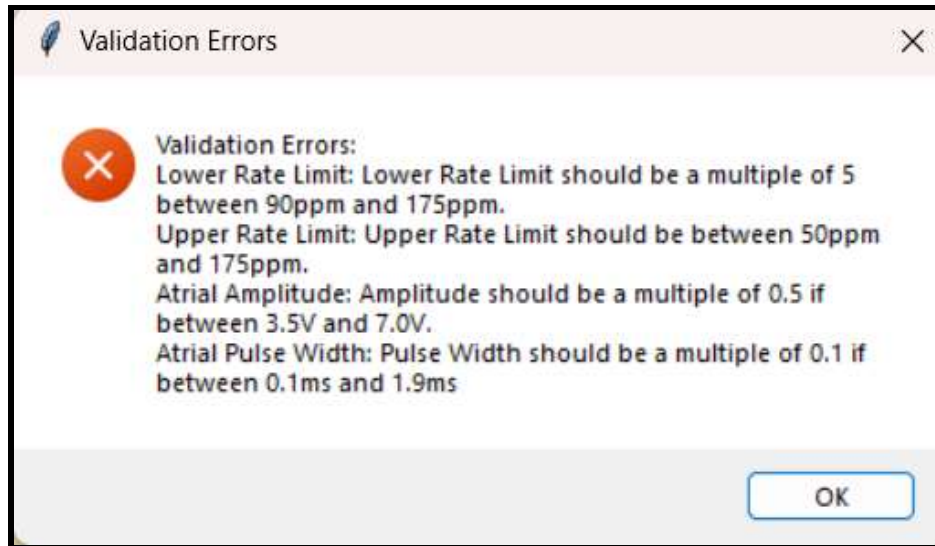


Figure 55: Message Box Implementation for Invalid Parameter Input

However, during our process of creating these pop ups, we realized that our message pop-ups did not function the way we wanted them to function. For example, if there were an invalid entry within an entry box and while not clicking off the entry box, if you were to submit, the submission would accept the values that were present within the box prior to editing the specific entry box. This was not ideal as the user may think that the system had accepted their invalid entry. Thus, we created a function called **validate_parameter** which would use the `get()` function in order to extract the value that is within the entry as of that moment. After, it would pass that value into a function called **validate_parameter_val** which would essentially check what the parameter for the corresponding value is and check its validity. This allowed us to have the error pop-ups appear according to what is within the entry boxes at the moment instead of what is within the entry boxes after the user clicks away from the entry box.

Another roadblock we faced in this section was that current values in the entries wouldn't save without focusing out if submit was clicked (only saved upon double click of submit). We fixed it by making a new list to store current entry values by using `get()`. Then we used the `self.mode_parameter(self.current_mode)` in order to create a zip list within the `show_parameter_values`. This allowed us to create a list of current parameters and their corresponding values which we later used within the `store_parameter_values` function

Additionally, if we clicked into another mode without pressing next, and then directly submitted, it would try to store values for entries that weren't rendered on the screen yet which could do funky things like put hysteresis as a value for the key of AOO. We fixed it by using another variable which would store the current mode. This was done by updating the current mode variable after pressing next which directs the user into the render function. Thus, we had the string of `self.mode_var.get()` equal to the current mode variable only when the render function was called. Now, using this new variable, we checked whether the current mode variable and the `mode_var` variable were equal. This allowed us to implement a conditional, where it would throw an error if the modes are not equal. However, if it is equal, the code would run normally which fixed the last error we faced in this section. We just added route buttons for the next part of the process (display existing data and egram).

Figure 56: Final GUI for Mode Selection with all Elements

Displaying Existing Data

After adding the route buttons shown in Figure 18, we created a new module for displaying the existing data on a new frame called **display.py**. This module was the simplest to implement as all we had to do was filter through our `pacemaker_data.json` and display values for a given key. The first step was to iterate through the JSON file and get all the keys which were the pacemaker numbers and add them to a dropdown list. When the user selects a pacemaker and presses 'Display', the value for the key selected in the dropdown would be displayed. We faced extremely minimal error handling for this module since we had pretty much recycled our JSON functions from the previous modules.

Figure 57: GUI for the Display Existing Data Frame

Placeholder for Electrogram Data

The last step for us was to implement a placeholder for the electrogram data to make it easier to integrate when we received the data. As such, we decided to use the matplotlib library of Python to display an empty graph in the middle of the screen with labelled axes.

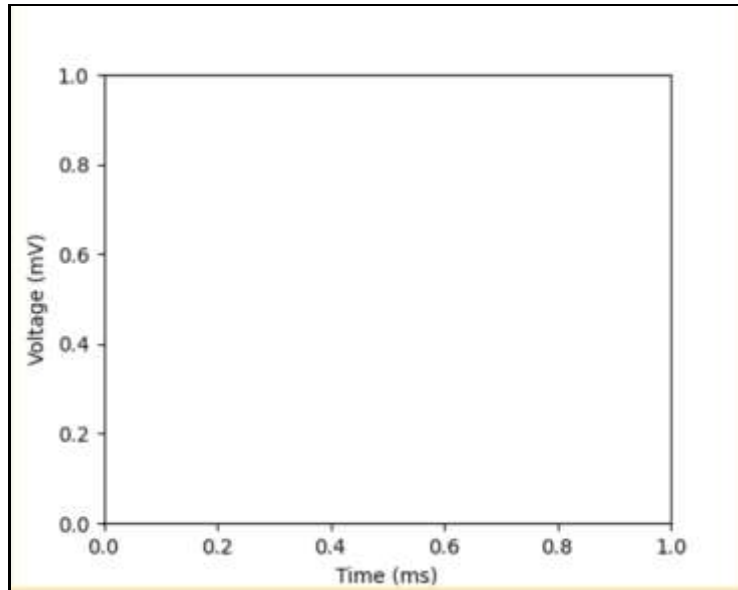


Figure 58: Electrogram Data Placeholder Graph

The modifications for Assignment 1 conclude here, assisting in the scope of Assignment 2. Initially, our attention was directed towards refining the user experience. We addressed an outstanding issue from Assignment 1 by integrating the username with the pacemaker data entries. This adjustment ensured that the `mode_sel.py` module reset appropriately when the users navigated back to the pacemaker screen. Subsequently, we expanded the functionality by incorporating new modes (AOOR, VOOR, AAIR, VVIR) and their associated parameters. Each parameter required its validation functions, laying the foundation for the complexities of Assignment 2. Serial communication was a pivotal facet of Assignment 2. A module was created for it called `serial_communication.py`, initially focusing on configuring the Data Communication Module (DCM) to verify pacemaker connectivity. This module prevented the `mole_sel.py` module from resetting unless a physical connection was established upon returning to the pacemaker screen. Further refinement involved removing the redundant PVARP parameter. The subsequent step centered on preparing to transmit data to the FRDM K64 board using the UART protocol. This streamlined the packet by omitting variables such as AV Delay, Hysteresis, and Rate Smoothing, aligning with the Simulink team's specifications. The development of a customized serial packet protocol ensured, facilitating seamless data exchange. Despite encountering challenges during testing, the team succeeded in ensuring the integrity of transmitted data to the pacemaker. An innovative feature was introduced, enabling the detection of an entirely new physical board by parsing the serial number from the hardware ID array. Functions from the `serial_communication.py` module were encapsulated within a class, aligning with existing data storage methods. This laid the groundwork for programming the electrogram (egram) segment of the DCM. This phase involved continuous reception of two doubles representing atrial and ventricular signals, presented on a plot for comprehensive visualization.

Further Improvements

During our next phase we had realized that there were some issues that we had not mentioned within our code. One of these issues was a privacy issue, where when a user logout, the entries which the previous user had entered wouldn't clear out from the entry boxes for the next user. In addition, the previous screen which the previous user was using would be left as it is for the next user including the mode which they were using. This was a huge issue as this would invade the privacy of other users. The way we fixed this issue is by resetting our mode selection screen each time a user would either go back or log out of their account. This was an easy implementation as what we had done was creating a `reset_mode_sel` function which would reset the mode selected back to a blank string as well as deleting the current error labels by checking whether there is any existing error labels present on the screen and in addition, all the current widgets would be deleted. This function is ran each time the back or the logout button is pressed from the mode selection screen, which allows the current user and the new users have access to a whole new mode selection screen.

```
def reset_mode_sel(self):
    self.mode_var.set("")
    if self.error_labels.items() != {}:
        for error_label in self.error_labels.items():
            error_label[1].config(text="")
    for widget in self.current_widgets:
        widget.destroy()
```

Figure 59: `reset_mode_sel` function

Our next issue that we had to fix was an issue regarding our display screen as well as the way we had stored our user data. This issue had shown us a huge error which we had not realized which was the fact that the pacemaker data was not entered into our JSON database according to each unique user. For example, if one user accesses pacemaker named "pacemaker 1", when another user would sign in, they would be able to display the information which the other user had submitted for "pacemaker 1". This was a naïve mistake which our group had not realized prior to creating our display screen. In order to fix this issue, what we had done was to store every piece of submission under the user who is signed in. Thus, whenever a user would sign in, the data which they would submit would store within our database as shown below.

{username: {Pacemaker 1: {Mode: {parameters}}, Mode: {parameters}}, Pacemaker 2: {Mode: {parameters}}, Mode: {parameters}}}

We had also implemented this towards how we store the previous used pacemaker by storing our data within our JSON database in the following form which allowed us to display the previous pacemaker which was connected prior for the signed in user instead of the previous pacemaker connected to the general program.

{username 1: Previous pacemaker, username 2: Previous pacemaker, username 3: Previous pacemaker}

While addressing these issues, we had also realized that our display screen would also need to change since it had lacked a clean and clear visual for the user to be able to distinguish with all this data being displayed as it is straight from the database in the form of {username: {Pacemaker 1: {Mode:

{parameters}, Mode: {parameters}}, Pacemaker 2: {Mode: {parameters}, Mode: {parameters}}}. This was not ideal as this was not very user friendly as this was visually difficult for the user to distinguish their desired data as they would like. Thus we made a design decision to change this feature by displaying our data in a table pattern which would display each Mode, parameter and their respective values in a straight column. This would extend to the right as the user would submit more and more values. We soon faced a problem with this idea as we had realized that we would not be able to display all the modes within the visible screen as it simply too much. Thus we introduced a scroll bar for the user below the display canvas which would allow the user to scroll through the data as they would like.



Figure 60: New Display Screen

User-specific functionality is introduced by loading the pacemaker data based on the username. This change indicates a shift towards a more personalized experience, where data is filtered based on the logged-in user every time. The visual layout and style of UI elements have been adjusted. For instance, the background color (self.main.bgcolor2) is now a parameter, indicating a cohesive color scheme across the application. Introduction of new labels, and modification in the placement and configuration of existing for clearer and more organized display of information.

A significant enhancement is the use of a canvas and a frame within it to display pacemaker data. This approach allows for more data to be displayed in an organized manner, especially when dealing with large datasets. The addition of a scrollbar self.scrollbar to the canvas improves the user interface by allowing users to scroll through the data, making the application more user-friendly for extensive data sets. The method load_existing_pacemakers has been updated to filter pacemakers based on the current username. This change enhances the application's security and privacy by ensuring users access only their data and none other's data. The method display_parameters now display data in a grid layout within the canvas, providing a more structured and readable format for presenting pacemaker parameters and its data. This method checks if there are pacemakers associated with the user and it then switches the displayed frame accordingly. This change allows for a more dynamic and responsive UI, adapting to the available data.

Changes Made To user_manager.py

```
class UserManager:
    def __init__(self, user_data_file, main_app):
        self.user_data_file = user_data_file
        self.users = self.load_user_data()
        self.main = main_app
        self.serial_comm = SerialCommunication(self.main)

    def load_user_data(self):
        try:
            with open(self.user_data_file, "r") as file:
                return json.load(file)
        except FileNotFoundError:
            return []

    def save_user_data(self):
        with open(self.user_data_file, "w") as file:
            json.dump(self.users, file)

    def register_user(self, username, password, confirm):
        if len(self.users) >= 3:
            return messagebox.showerror("Registration Error", "Maximum 10 users reached.")

        for user_data in self.users:
            if username == user_data['username']:
                return messagebox.showerror("Registration Error", "This username is already in use")

        if username and password:
            if(password == confirm):
                self.users.append({"username": username, "password": password})
                self.save_user_data()
                return messagebox.showinfo("Registration", "User registered successfully!")
            else:
                return messagebox.showerror("Registration Error", "Passwords don't match!")
        else:
            return messagebox.showerror("Registration Error", "Username and password are required.")

    def login_user(self, username, password):
        for user_data in self.users:
            if username == user_data["username"] and password == user_data["password"]:
                self.main.login_bool = False
                self.main.port_list = ["", ""]
                self.main.route(self.main.pacemaker_sel)
                self.main.pacemaker_interface.check_and_update_connection()
                return messagebox.showinfo("Login", "Login successful!")

        return messagebox.showerror("Login Error", "Invalid username or password.")
```

Figure 61: User Manager Class

The modifications made to this module extends its capabilities to integrate with additional system components, notably the serial communication module and enhanced user session management. The updated version imports a module named SerialCommunication. This suggests an expansion of the module's functionality to include communication with external hardware, which in this case is a

pacemaker device. A new instance variable `self.serial_comm` is initialized in the constructor. This instance is created by passing `self.main`, which is a reference to the main application/module, to the `SerialCommunication` class. This change indicates that the user management module now has direct access to serial communication functionalities, allowing it to perform tasks like sending or receiving data to/from the hardware, which is a pacemaker as part of user-specific operations.

In the original version, the `login_user` method simply routed to the pacemaker selection frame upon successful login. Well in the updated version, additional steps are performed upon successful login such as `self.main.login_bool` is set to `False`, which serves as a flag for letting the software know whether the user is on the login frame or not.

Changes Made To `pacemaker_interface.py`

A pivotal enhancement in the new version of the module is the integration with the `SerialCommunication` class from the `serial_communication` module. This integration is a substantial leap towards increasing the real-world applicability of the software. By enabling direct communication with the pacemaker device, the application moves from a theoretical or simulated environment to one capable of actual device interaction. This change marks a significant step forward in the software's evolution, transforming it into a more practical tool for pacemaker configuration and management.

This module code introduces two new methods, `update_connection_label` and `check_and_update_connection`, which collectively enhance the real-time feedback provided to the user regarding the pacemaker's connection status. The `update_connection_label` method dynamically alters the text and color of the connection status label to reflect the current state of connectivity with the pacemaker. This method likely relies on the new `SerialCommunication` integration to determine the actual connection status. The `check_and_update_connection` method utilizes the Tkinter `after` method to periodically invoke the update function. This ensures that the user is constantly informed of the connection status in real-time, thereby enhancing the application's responsiveness and user experience.

The modifications in data handling are a notable improvement in the module. The application now utilizes a `username` variable to personalize the loading and saving of pacemaker information. This variable, presumably obtained from the main application (`main_app`), allows the software to provide to individual user needs by maintaining separate configurations for different users. This approach enhances the application's security and personalization, ensuring that users have access to their specific settings and history, thereby significantly improving user experience and data confidentiality.

The updated submit method demonstrates a more rigorous approach to validation and error handling. Before proceeding with the submission, the method now checks for an actual connection with the pacemaker using the new `serial_comm` integration. This practical validation ensures that the user's interaction with the application is not only theoretical but also grounded in real-world applicability. Additionally, the method includes robust checks for blank or inappropriate pacemaker names, thereby maintaining the integrity and reliability of user inputs.

Alongside functional enhancements, the update has brought about several visual and textual improvements. Changes in the text of labels and background colors are evident. For instance, the pacemaker entry label's text is now "Insert name for Pacemaker", and the background color is set to a different colour. These changes are likely intended to improve the visual consistency and appeal of the user interface, making the application more engaging and user-friendly.

Addition of New Modes & Changes Made To `mode_sel.py`

Prior to creating our communication path for our pacemaker and heart into our DCM, we had some new mode additions which we needed to consider into our software which were fairly easy to implement as we had created a structure which is easy to adjust if needed during our starting stage. This allowed a smooth and quick implementation of our new 4 modes.



Figure 62: New Modes Shown in the Dropdown Menu

Several significant changes and enhancements were made which were introduction of the new modes, additional parameters for new modes, inclusion of serial communication, nominal values dictionary, enhanced validation functions, modified the existing amplitude and pulse width validation, user-specific data storage, data sending to the pacemaker, user interface enhancements, and improved error handling and user feedback.

Four additional pacing modes were added: AOOR, VOOR, AAIR, and VVIR. This expansion significantly broadened the application's scope, allowing it to serve to more sophisticated pacemaker functionalities. These new modes are crucial in modern pacemakers as they offer advanced pacing options, addressing a wider range of cardiac conditions. The inclusion of these modes demonstrates an effort to keep the application aligned with contemporary medical standards and practices in cardiac pacing in the real world. With the new pacing modes, the application now encompasses additional parameters such as "Maximum Sensor Rate", "Activity Threshold", "Reaction Time", "Response Factor", and "Recovery Time". These parameters are essential to the rate-adaptive pacing modes, which are designed to adjust the pacing rate in response to physiological demands, such as during physical activity (running, jogging, walking). This enhancement is a significant step towards making the application more relevant and useful in a clinical setting, where personalized and adaptive pacemaker settings are crucial for patient-specific treatments.

The introduction of `self.serial_comm` and the integration with the `SerialCommunication` class is a pivotal update. This feature overpasses the gap between the user interface and the actual pacemaker hardware, allowing for direct communication and real-time updating of the pacemaker settings. This addition marks a transition towards a more practical and hands-on application, capable of directly influencing pacemaker behavior, thereby enhancing its utility in the real-world clinical scenarios. The

new self.nominal_values dictionary was a necessary addition to this module, providing a set of default or 'nominal' values for various pacemaker parameters. This feature is particularly useful for quickly resetting parameters to standard values which are the nominal values, essential during initial device setup or when returning to baseline settings. It reflects a user-centric approach, simplifying the process of managing and adjusting pacemaker settings. The expanded range of parameters involved the development of additional validation functions. These functions ensure that inputs for parameters like maximum sensor rate, activity threshold, reaction time, response factor, and recovery time adhere to the expected standards.

A critical update in the show_parameter_values method is the capability to send configured parameters directly to the pacemaker. This functionality closes the loop between user input and hardware action, allowing for the immediate implementation of chosen settings. These improvements demonstrate a commitment to user safety and a seamless user experience, ensuring that the application is not only functional but also reliable and easy to use. Also, to prevent empty JSON errors, we had plenty of error handling as shown below:

```
def store_parameter_values(self):
    # Load existing JSON data
    try:
        with open("DCM/DataStorage/pacemaker_data.json", "r") as file:
            try:
                json_data = json.load(file)
            except json.decoder.JSONDecodeError:
                # Handle the case of an empty JSON file
                json_data = {}
        except FileNotFoundError:
            json_data = {}
    if not json_data:
        json_data = {}
```

Figure 63: Opening JSON with Error Handling

Serial Communication

This was one of the modules which we had given much attention to during our second phase of this project. This module ensured the communication of the pacemaker and heart to our DCM which allowed us to allow the user to observe certain data that would be output from the pacemaker as well as the heart. This module mainly uses PySerial as a way of communicating with the pacemaker as well as the heart which would either read data from the pacemaker in order to display data for the user such as displaying Egram data or write data into the pacemaker in order to implement certain modes and receive desired results from the pacemaker.

During the beginning, our team was focused on how to distinguish between the two FRDM K64 boards (pacemaker and heart). This was a challenge simply due to the nature of our problem where the ports which would connect for each of our computers were different. After some thorough experimenting with our fellow colleagues, our team had realized that the hardware ID for both the pacemaker and the heart are two different IDs which do not change from device to device.

```
USB VID:PID=1366:1015 SER=000621000000 - PACEMAKER HWID
USB VID:PID=0483:374B SER=0667FF313736504157096040 LOCATION=1-1:x.2 - HEART HWID
```

This worked in our favour, as this gave us a way of distinguishing our two boards from one and another. Thus we had used the command `serial.tools.list_ports.comports()` in order to access a list of devices that are connected to our device. This list would contain three objects within called ports, showing which physical port that the device is connected to, desc, showing a physical description of the device, and hwid, showing the corresponding device's hwid ordered as such. This allowed us to iterate through this list in order find our corresponding hwid's and if the hwid which we are looking for matches to that of the iteration, it would assign the corresponding port to a the variables `pacemaker_port` or `heart_port` depending on which device hwid it matches with. We had tested this with multiple devices and it proved to show no errors during our test stage. This also allowed us to create a function which checks whether the pacemaker is connected to our DCM by returning a variable named `connected` as true if the hwid for the pacemaker is found and false if not found. This function is ran every 0.1 seconds in order to show an accurate and clear depiction of the pacemaker connectivity. After tracking this data, we now have 4 different json files for 4 unique purposes as shown in Figure 64 below.

- 1) `user_data.json` for storing username/password
- 2) `pacemaker_data.json` for storing user specific programmable data
- 3) `pacemaker_board_list.json` for storing data specific to each board
- 4) `previous_pacemaker.json` for storing user specific previous pacemaker states

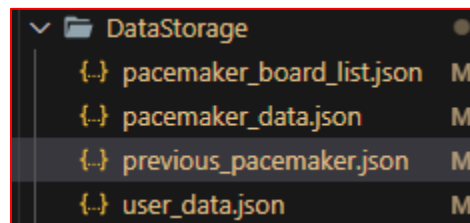


Figure 64: All Data Storage Files Used

The next stage was for us to establish and check whether the communication with the pacemaker was functioning as we intended. This involved testing the sending and receiving of the data. The figure below demonstrates how we are sending data to the pacemaker.

```

packet = []

s2 = struct.pack('B', data_to_send['MODE'])
s3 = struct.pack('H', data_to_send['LRL'])
s4 = struct.pack('H', data_to_send['URL'])
s5 = struct.pack('H', data_to_send['MSR'])
s6 = struct.pack('f', data_to_send['A_AMPLITUDE'])
s7 = struct.pack('f', data_to_send['V_AMPLITUDE'])
s8 = struct.pack('H', data_to_send['A_WIDTH'])
s9 = struct.pack('H', data_to_send['V_WIDTH'])
s10 = struct.pack('f', data_to_send['A_SENSITIVITY'])
s11 = struct.pack('f', data_to_send['V_SENSITIVITY'])
s12 = struct.pack('H', data_to_send['VRP'])
s13 = struct.pack('H', data_to_send['ARP'])
s14 = struct.pack('B', activity_thresh_value)
s15 = struct.pack('H', data_to_send['REACT_TIME'])
s16 = struct.pack('H', data_to_send['RESPONSE_FAC'])
s17 = struct.pack('H', data_to_send['RECOVERY_TIME'])

packet.append(s0)
packet.append(s1)
packet.append(s2)
packet.append(s3)
packet.append(s4)
packet.append(s5)
packet.append(s6)
packet.append(s7)
packet.append(s8)
packet.append(s9)
packet.append(s10)
packet.append(s11)
packet.append(s12)
packet.append(s13)
packet.append(s14)
packet.append(s15)
packet.append(s16)
packet.append(s17)

#Establish Serial Connection
try:
    ser = serial.Serial(self.main.pacemaker_port, 115200)
except:
    messagebox.showerror("Error", "Please check your Pacemaker connection before starting Egram")

ser.write(b''.join(packet))

```

Figure 65: Sending Data Packet

Then, we receive the data as shown below:

```

#Unpacking Params
model = (struct.unpack('B',ser.read(1)))[0]
lrl = (struct.unpack('H',ser.read(2)))[0]
url = (struct.unpack('H',ser.read(2)))[0]
msr = (struct.unpack('H',ser.read(2)))[0]
a_amplitude = (struct.unpack('f',ser.read(4)))[0]
v_amplitude = (struct.unpack('f',ser.read(4)))[0]
a_width = (struct.unpack('H',ser.read(2)))[0]
v_width = (struct.unpack('H',ser.read(2)))[0]
a_sensitivity = (struct.unpack('f',ser.read(4)))[0]
v_sensitivity = (struct.unpack('f',ser.read(4)))[0]
vrp = (struct.unpack('H',ser.read(2)))[0]
arp = (struct.unpack('H',ser.read(2)))[0]
activity_thresh = (struct.unpack('B',ser.read(1)))[0]
react_time = (struct.unpack('H',ser.read(2)))[0]
r_factor = (struct.unpack('H',ser.read(2)))[0]
rec_time = (struct.unpack('H',ser.read(2)))[0]
a_signal = (struct.unpack('d',ser.read(8)))[0]
v_signal = (struct.unpack('d',ser.read(8)))[0]
receivedarray = [model,lrl,url,msr,a_amplitude,v_amplitude,a_width,v_width, a_sensitivity, v_sensitivity,vrp,arp,activity_thresh,react_time,r_factor,rec_time]
egram_data = [a_signal,v_signal]

```

Figure 66: Unpacking Received Data

```

error = 0
while(error == 0):
    if(data_to_send['MODE'] != round(modeN)):
        error = 1
    elif(data_to_send['LRL'] != round(lrl)):
        error = 1
    elif(data_to_send['URL'] != round(url)):
        error = 1
    elif(data_to_send['MSR'] != round(msr)):
        error = 1
    elif(data_to_send['A_AMPLITUDE'] != round(a_amplitude,1)):
        error = 1
    elif(data_to_send['V_AMPLITUDE'] != round(v_amplitude,1)):
        error = 1
    elif(data_to_send['A_WIDTH'] != round(a_width,1)):
        error = 1
    elif(data_to_send['V_WIDTH'] != round(v_width,1)):
        error = 1
    elif(data_to_send['A_SENSITIVITY'] != round(a_sensitivity)):
        error = 1
    elif(data_to_send['V_SENSITIVITY'] != round(v_sensitivity)):
        error = 1
    elif(data_to_send['VRP'] != round(vrp)):
        error = 1
    elif(data_to_send['ARP'] != round(arp)):
        error = 1
    elif(activity_thresh_value != round(activity_thresh)):
        error = 1
    elif(data_to_send['REACT_TIME'] != round(react_time)):
        error = 1
    elif(data_to_send['RESPONSE_FAC'] != round(r_factor)):
        error = 1
    elif(data_to_send['RECOVERY_TIME'] != round(rec_time)):
        error = 1
    else:
        error = 2

if(error == 1):
    if(s1 != b'\x02'):
        messagebox.showinfo("Note!", "There was a problem communicating with the Pacemaker")
    else:
        messagebox.showinfo("Note!", "The parameters have been confirmed with the Pacemaker")
ser.close() # Close the serial connection after sending

return egram_data

```

Figure 67: Verification of Received Data

In order to implement this, we had added a new command to our submit parameters button which would also transfer the values submitted into the serial communication module which would later write

these variables into the pacemaker using the communication which we had established using PySerial. Then we would receive the data packet and unpack it to extract the necessary data including the egram data. Our assurance case diagram explains this.

This module can identify specific hardware devices by their unique identifiers, ensuring that the correct devices are being communicated with. It maintains a record of devices for each user, helping in personalizing the application experience. Converts pacemaker parameters into a specific binary format suitable for serial communication and sends them to the device, which is the pacemaker. After sending parameters, it reads back the data from the pacemaker to ensure that the correct settings have been applied. The class includes try-except blocks for error handling during file operations and serial communication, enhancing the robustness of the application. It uses message boxes to provide feedback to the user about the status of device connections and parameter transmissions. External libraries were used in this module as well such as serial library, used for handling serial communication with external devices. json library, employed for reading and writing JSON files, which are used for storing user-specific device information. struct library, utilized for packing and unpacking data into a binary format which is suitable for serial transmission. tkinter.messagebox module, which provides a simple way to display warning and error messages to the user.

Future Requirement Changes

Anticipating future requirement changes is crucial for ensuring the adaptability and sustainability of our application. As we prepare to implement additional pacing modes, such as DOO, and DOOR, it's important to maintain information hiding and low coupling in our modules, allowing for efficient integration of these new features without disrupting the existing functionality of our code. Though it may not be a requirement, if we were to transition from JSON files to another database, this should be executed with a focus on high cohesion to ensure that related data elements are stored together logically. User interface enhancements according to the user feedback should be pursued while preserving low coupling to the backend, ensuring that any adjustments are seamlessly integrated. For testing and validation, accommodating new test cases and complying with new updated standards is essential, and maintaining information hiding will help isolate changes to the testing modules. With potential changes in security and compliance regulations, our system must prioritize information hiding and low coupling to allow for easy adaptation while keeping sensitive data secured. Lastly, any performance improvements require a high focus on high cohesion in algorithms and data structures to enhance efficiency without causing unintended side effects in other parts of the system. Lastly, these principles of information hiding, low coupling, and high cohesion will be very important in shaping our application to meet future requirements effectively and sustainably.

Future Design Changes

Currently, JSON files are used to store data locally. While this works functionally, there may be risk with information hiding since the security of a JSON file is comparable to that of a text file. This can be improved with the use of a database that requires a password to enter. Encryption is outside the scope of this course but using a simple database with SQL which would require queries to access data is already much safer than a human-readable format JSON file. As the number of pacing modes increase, it may be useful to modularize our validation methods from mode_sel.py into another module called validation.py since that would prevent the module from becoming too difficult to debug while maintaining high

cohesion. We may need to provide the option to delete the users manually through the GUI but maybe restrict it to users with admin status. Since we don't have an admin feature included in our registration process, that would be another design change we would have to implement. The electrogram is not able to continuously update without crashing, so in the future we want to get rid off the time delay of 75ms and make it run fluently to get better pulses. Another concern we currently have is our 'Display Existing Data' displays data entered on a previous run of the application so if you just entered a set of values and then clicked the button, it will not show the updated values for that pacemaker until you restart the app. We will need to change this to make testing easier and avoid confusing the user into thinking that they have not successfully saved the data since they don't have access to the JSON files. We also need to add more private functions instead of using public functions to add more information hiding.

Explanation of Modules with Testing

The goal of this section is to explain the general purpose of each module as well as the black box function of these modules. We will go more into depth into each module where we will explain what each function does as well as the testing stages that we had went through in order to perfect our modules.

`mode_sel.py`:

Purpose:

The purpose of the `Mode_sel.py` module is to control the modes of the GUI with error messages of invalid information, store data in a JSON file and provide access to buttons that serve other purposes such as displaying previous data and viewing electrogram data. However, in general, this code is part of a user interface application that allows users to select a specific "mode" such as "AOO", "VOO", "AAI," or "VVI". For each selected mode, the user can adjust several parameters related to that mode, such as rate limit, amplitudes, pulse width, and so on. The module provides a simple user interface for entering these parameters and performs real-time verification to ensure that the entered values meet the requirements of the given ranges and displays error messages to notify the user if there are any validation errors. If the user sets the selected modality parameters correctly, the application allows them to save these settings for the specific pacemaker device.

Public Functions and Black box behaviour:

`__init__(self, root, main_app)`: This function lays out the physical appearance of the mode selection screen prior to selecting any mode. It lays down many visual aspects of the mode screen from buttons to labels. It also initializes dictionaries to store error labels and mode parameters. Additionally, it defines the default values for pacemaker parameters and sets up event handling for input validation.

`render(self, mode)`: This function renders the corresponding entries and parameters according to the mode that the user has selected. Depending on the selected mode, it dynamically creates labels, entry widgets, and error labels for relevant parameters while clearing any previous widgets. Users can input parameter values, and event handling is set up to validate and update the parameter values accordingly.

`validate_lower_rate_limit(self, value)`: This function ensures that the lower rate limit is within a valid range and follows specific rules based on the numeric range.

validate_upper_rate_limit(self, value): This function ensures that the upper rate limit is within a valid range and follows specific rules based on the numeric range.

validate_amplitude(self, value): This function ensures that the atrial and ventricular amplitudes are within a valid range and follow specific rules based on the numeric range.

validate_pulse_width(self, value): This function ensures that the atrial and ventricular pulse widths are within a valid range and follow specific rules based on the numeric range.

validate_sensitivity(self, value): This function ensures that the atrial and ventricular sensitivities are within a valid range and follow specific rules based on the numeric range.

validate_refractory_period(self, value): This function ensures that the atrial and ventricular refractory periods are within a valid range and follow specific rules based on the numeric range.

validate_hysteresis(self, value): This function ensures that the hysteresis is within a valid range and follows specific rules based on the numeric range.

validate_rate_smoothing(self, value): This function ensures that the rate smoothing is within a valid range and follows specific rules based on the numeric range.

update_parameter_value(self, entry, param): This function updates the parameter value when an entry widget loses focus. Depending on the parameter type, it triggers the corresponding validation function and updates the parameter value or displays an error message in the associated error label.

store_parameter_values(self): This function is responsible for storing the parameter values in a JSON file thus managing the persistence of parameter values for different pacemakers and modes.

show_parameter_values(self): It first validates the parameter values, and if there are any errors, it displays a message box with validation error messages. If all values are valid, it calls `store_parameter_values` to store the parameters and displays a success message in a message box.

validate_parameters(self): This function essentially collects all validation errors for each parameter if one exists and It returns a dictionary of validation errors for further processing.

validate_parameter_val(self, param, value): This function makes sure that the value for a parameter is either valid or not. If it is not valid, it will send the corresponding error message as a string. If it is valid, it will send a "Valid" string.

Global Variables:

Self.error_labels: This is a list of all error labels meant to be placed in front of parameters when the user focuses out of the entry box.

self.mode_label: This is a label saying "Select Mode" over the drop down menu in order show the user

self.mode_var: This variable shows the mode that the interface is in as of that moment its called

self.log_out: This is a button for logging the user out of the page. It will redirect them on to the login page.

self.back_button: This is a button for going back. It will redirect them back to the pacemaker selection screen.

self.mode_parameters: This is a list of all modes with their corresponding parameters.

self.current_widgets: This is essentially list of all widgets that are present on the screen as of that moment (it resets each time the next button is clicked and is updated with the new widgets)

self.current_vals: This is a list of all the values from the entries that are present on the screen as of the moment which were obtained using get() therefore they are the live values of the entries

self.current_param_vals: This is a list of the current parameters and entries which are on the screen as of that moment.

self.current_mode: This variable is the current mode which is on display on the mode selection screen. This variable only gets updated when next is pressed on the mode selection interface.

self.parameter_values: This is a list of all the parameters with their corresponding nominal values.

Private Functions:

This module does not contain any private functions.

Testing and Results:

Test 1 (Failed - Required Fix):

Error labels: During our stage of making the error labels in front of the parameters, we went through every parameter and entered an invalid number onto the entry to see whether the error label would pop up when we were focused out of the entry box.



Figure 68: Error Label not Showing for Upper Rate Limit

During the first stage, some of the boxes were **not displaying** the error message due to the way that conditionals were placed. However, after adjusting the conditionals to an if, elif, elif... system, we

got the labels to work. In addition, we had trouble with entries such as off with certain parameters. Therefore, decided to use 0 as off for simplicity as well as for the ease future tasks such as graphing. In addition, we tested out multiple valid values in order to determine whether the labels were cleared whenever there was a valid value within the entry box.

Test 2 (Failed - Required Fix):

Python arithmetic errors with Modulus: We tested the validity of our parameters by inserting all the values for each parameter into the entry box and we checked for the error labels in front of the parameter in order to determine the accuracy of the functions which check our value's validity. During our test stage, we realized that certain values which were correct for the parameters atrial pulse width, ventricular pulse width, ventricular amplitude, and atrial amplitude had shown error labels in front of the parameters which was **not supposed to happen**.

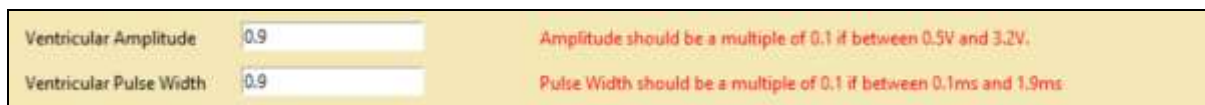


Figure 69: Error Label Malfunctioning Due to Modulus Logic

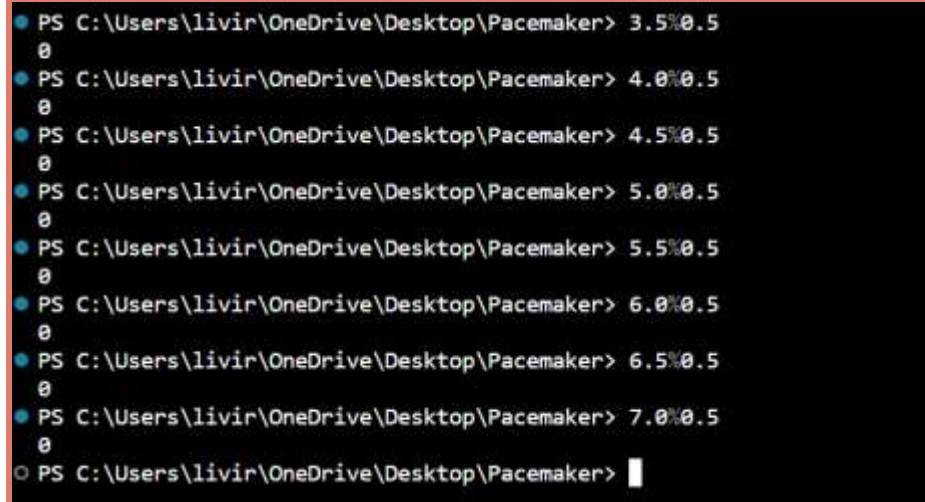
Therefore, we found an error within our code. We had checked the syntax as well as the logic of our code thoroughly and no errors were to be found. However, later we suspected the function of the modulus operator, thus we performed the test below.

```
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.3%0.1
0.1
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.1%0.1
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.2%0.1
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.3%0.1
0.1
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.4%0.1
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.5%0.1
0.1
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.6%0.1
0.1
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.7%0.1
0.09999999999999999
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.8%0.1
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 0.9%0.1
0.1
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 1.0%0.1
0.1
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 1.1%0.1
2.77555756156289E-17
```

Figure 70: Test 1 of modulus with 0.1

As you are able to see, the modulus operator has certain values which do not display its correct modulus value and you are able to see within the image that the accuracy of the modulus operation is not consistent in any way. We suspected this was due to rounding errors. However, this error only occurs when the modulus is of division by 0.1 or a very small number. We tested this for other numbers such as 0.5 and we weren't able to find any error with the modulus operator with this test. Thus, we kept the

modulus for values greater than 0.5 and **fixed** the rest by using an iterative approach described in our process.



```
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 3.5%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 4.0%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 4.5%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 5.0%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 5.5%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 6.0%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 6.5%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 7.0%0.5
0
PS C:\Users\livir\OneDrive\Desktop\Pacemaker> 
```

Figure 71: Test 2 of modulus with 0.5

Test 3 (Failed - Required Fix):

Error popups: During this stage, we tested this feature out by essentially submitting parameters. First, we submitted valid parameters in order to see whether the right pop-ups were being shown as well as the right data was being recorded on the JSON file. Then we submitted invalid parameters in order to see whether the error pop-ups, as well as whether the invalid values were going through into the JSON file. During the third stage, we tested whether both stages 1 and 2 when the modes switched after a successful submission. During our testing, stage 1 and 2 did **not work** at first as it was taking the values of the entries from the update parameter values which only records the values of the entry boxes when the user clicks away. Therefore, when the user is still in the box and clicks submit, it wouldn't submit the value that they wanted to record instead the value that was within the box prior to that. We **fixed** this problem using get() function where we would get the value that is within the entry as of that moment whenever submit is pressed. Thus that problem was resolved. Then the next issue with stage 3, where after switching modes, we would get an error message whenever we clicked the submit button. This was due to the fact we were using the previous version of self.current_widgets where it **wouldn't reset** each time we would click next. Thus, after we made the code reset this list each time the next button is pressed, the list would only update to the entries that are on the screen as of that moment (including other widgets, however, we **solved** this problem by skipping every 2nd element which was always a label) which allowed us to fix that problem.

Test 4 (Passes):

In this test, we thoroughly **checked every parameter's error handling** and tried to get unwanted behavior but we were not able to get undesired behavior. We tried boundary/edge cases for each of the parameter values as well as testing unreasonable values of all sorts (such as extremely large/small numbers or numbers within the range with too much precision). So after all the test cases above, that failed at first and needed to be fixed. This was the only one where we **passed** it because at this point, we had dealt with almost all errors we could think of.

Lower Rate Limit	2345234	Lower Rate Limit should be between 30ppm and 175ppm.
Upper Rate Limit	234523	Upper Rate Limit should be between 50ppm and 175ppm.
Ventricular Amplitude	3.523453245	Amplitude should be a multiple of 0.5 if between 3.5V and 7.0V.
Ventricular Pulse Width	0.42345345	Pulse Width should be either 0.05ms or a multiple of 0.1 between 0.1ms and 1.9ms.
Ventricular Sensitivity	2.523546	Sensitivity should be a multiple of 0.5 if between 1mV and 10mV.
VRP	3206764	Refractory Period value should be between 150ms and 500ms.
Hysteresis	04567	Hysteresis should be 0ppm or between 30ppm and 175ppm.
Rate Smoothing	0345245	Rate Smoothing should be one of these values -> 0%, 3%, 6%, 9%, 12%, 15%, 18%, 21%, 25%

Figure 72: All error labels display

Test 5 (Passes):

In this test, we had thoroughly checked the error handling of every parameter and whether the intended error labels would show up in front of the parameters. In addition we had also checked whether the pop up error warning were functioning as intended as well.

Figure 73: Pop Up error warning functionality

Pacemaker Interface

Select Mode: AAR Next

View Egram Data Display Existing Data Back Log Out

Lower Rate Limit	60123	Lower Rate Limit should be between 30ppm and 175ppm.
Upper Rate Limit	3412	Upper Rate Limit should be between 50ppm and 175ppm.
Maximum Sensor Rate	2341	Maximum Sensor Rate should be between 50ppm and 175ppm.
Atrial Amplitude	2341	Amplitude should be 0V or between 0.7V - 3.0V.
Atrial Pulse Width	1341	Pulse Width should be a multiple of 1 between 3ms and 30ms.
Atrial Sensitivity	1341	Amplitude should be between 0V - 3V.
ARR	341	Refractory Period value should be a multiple of 10 between 150ms and 500ms.
Hysteresis	341	Hysteresis should be 0ppm or between 30ppm and 175ppm.
Rate Smoothing	3412	Rate Smoothing should be one of these values -> 0%, 2%, 6%, 8%, 12%, 15%, 18%, 21%, 25%.
Activity Threshold	431	Activity Threshold needs to be v-low, low, med-low, med, med-high, high or v-high.
Reaction Time	241	Reaction Time should be between 10sec and 50sec.
Response Factor	341	Response Factor should be a multiple of 1 between 1 and 16.
Recovery Time	341	

Submit Parameters

Figure 74: Error Label functionality

pacemaker_interface.py:

Purpose:

The primary purpose of the “pacemaker_interface.py” module is to provide a graphical user interface (GUI) for a pacemaker configuration application frame/screen. This module is responsible for displaying the previously connected pacemaker, which allows users to input the name of the pacemaker that they intend to connect, submitting the pacemaker’s name and facilitating the navigation to the next part of the application which is entering parameters for the four different modes. This module also offers the program to confirm with the user whether the entered pacemaker name differs from the previously connected pacemaker, and it displays a success message as well as it saves the information for future reference.

Public Functions:

def __init__(self, root, main_app): This is a constructor function that initializes the GUI interface. What it does is that it takes two parameters: root and main_app. The root parameter is a tkinter frame where main_app represents the main application. This function initializes the tkinter interface with the labels, entries, widgets, buttons, etc, and it also loads the previous pacemaker from the JSON file which is called “previous_pacemaker.json” and this is done using the “load_previous_pacemaker” function. This function also provides a GUI for users to input the name of the pacemaker that they want to connect to currently or it also gives them the option to continue with the pacemaker that was previously connected. It also deals with the button clicks events with the lambda functions, such as submitting the pacemaker information and logging out as well.

def load_precious_pacemaker(self): This function retrieves and loads the previously connected pacemaker’s name from a JSON file called “previous_pacemaker.json”. This function provides the user with the information about the previously connected pacemaker when they open the application. The function opens and reads the JSON file and if the file exists, it then extracts the value that is associated with it. It then returns the value in the JSON file, but if the file is not found, then it returns the default value which is “None” which indicates that there was no previous pacemaker.

def save_previous_pacemaker(self, pacemaker): This function updates and saves the name of the currently/newly connected pacemaker to the JSON file. This function is called when the user confirms connecting a different/new pacemaker. This function takes one parameter “pacemaker” which is just the name of the newly connected pacemaker. This gets created in then JSON file, where it overwrites the previous pacemaker’s name with the new pacemaker’s name, so in short it updates the newly connected pacemaker name from the previously connected pacemaker name.

def submit (self, entry): This function handles the pacemaker submission as well as the navigation to the next part/frame. This function takes one parameter “entry”, which is just the pacemaker’s name which is entered by the user. This function checks if the pacemaker’s name entered is different than the previously entered pacemaker name, if the pacemaker names are different, it asks the user confirmation which is done using the message box and if the user confirms it, it updates the previously pacemaker name to the name that was written. If the user does not confirm it, the previously pacemaker name says the same. A success message is displayed, and it then navigates to the next part of the application which is the frame to access the four different modes.

Global Variables:

self.prev_pacemaker: This global variable represents the previously connected pacemaker name and it stores it as a string

self.root: This global variable is a tkinter frame which describes as the root of the GUI

self.pacemaker_entry: This global variable is a tkinter entry widget where the users input the pacemaker names

self.previous_pacemaker_label: This global variable is a label widget which is used to display the previous pacemaker's name

self.pacemaker_label: This global variable is a label widget which is used for displaying the label "Pacemaker To Be Connected To"

self.connection_label: This global variable is a label widget which is used for displaying the status of the communication connection

self.submit_button: This global variable is a button widget which is used for submitting the pacemaker name

self.log_out: This global variable is a button widget for logging out

Private Functions:

This module does not contain any private functions.

Testing and Results:

Test 1 (Failed - Required Fix):

Displaying the previous pacemaker: During our stage of displaying the previous pacemaker that was connected, we kept going to the pacemaker interface to check if the previous pacemaker connected was updated or not. Initially, when we were trying to update/save the previous pacemaker which was connected, the pacemaker was not getting saved locally and it wasn't updating which returned "None" as shown below.

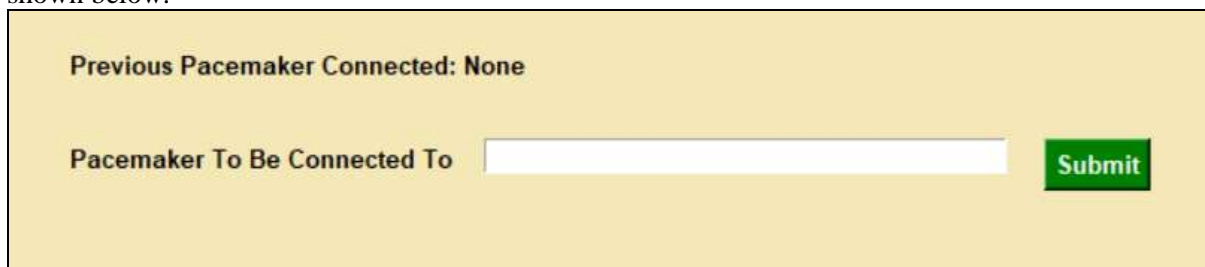


Figure 75: Previous Pacemaker Showing None (Not Saving)

To solve this issue, we create a JSON file to store the previously connected pacemaker names which then was able to save/update the precious pacemakers being connected and it was able to accurately display the precious pacemaker which was connected.



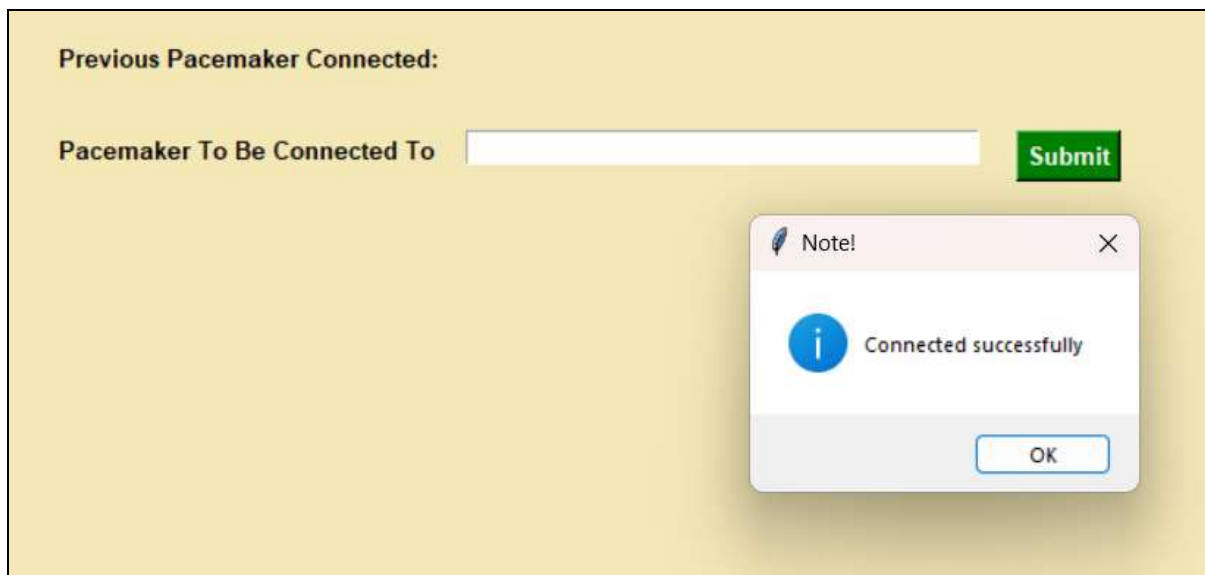
Previous Pacemaker Connected: 2

Pacemaker To Be Connected To

Figure 76: Previous Pacemaker Saving & Updating Label

Test 2 (Failed - Required Fix):

Blank Pacemaker Input: Testing for this was done by going into the Pacemaker Interface and not writing anything in the box of “Pacemaker To Be Connected To” and clicking submit which resulted in first saving it as a blank pacemaker.



Previous Pacemaker Connected:

Pacemaker To Be Connected To

Note!

Connected successfully

Figure 77: Not Throwing Error/Allowing Connection for Blank Pacemaker Input

This issue was fixed by adding an if statement saying that if the entry is empty, then it throws in an error message saying to enter a pacemaker, and after adding that if statement, this error was fixed as it prompted the user to put in an entry and this also stopped the saving of the blank pacemaker.

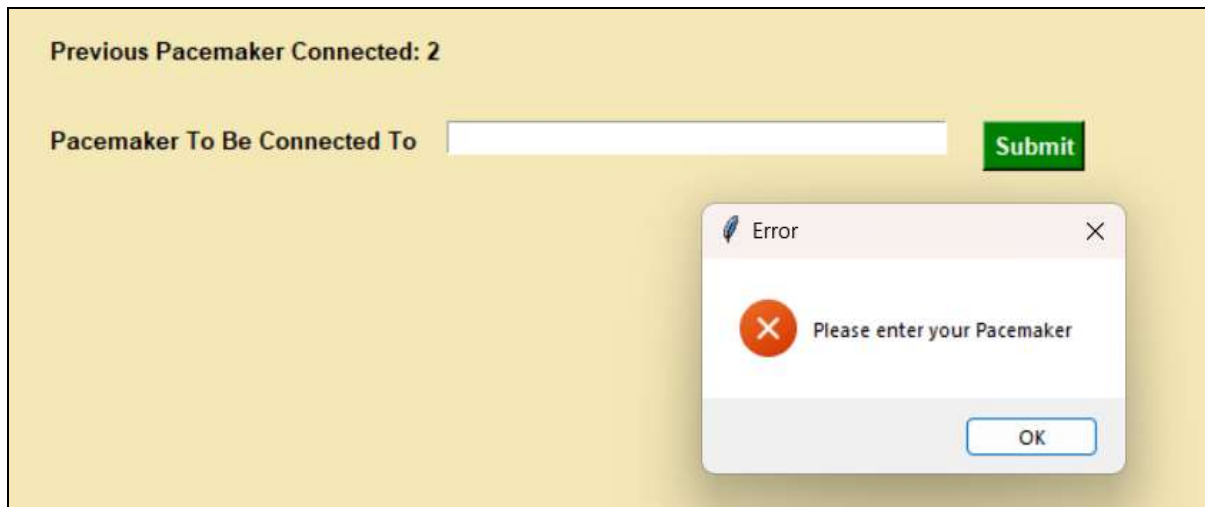


Figure 78: Fixed Error Message that blocks blank pacemaker input

Test 3 - (Failed - Required Fix):

Error with Blank Pacemaker Submission: When we added clearing of the entries upon leaving the page. Our code relied on reading the entry value to associate the parameter values entered with its corresponding pacemaker. As such, if we cleared the pacemaker entry, it would store the parameter values for a blank pacemaker inside the JSON file. To fix this, we had to remove the following line:

```
def route(self, target_frame):
    # Hide the current frame and show the target frame
    self.login_frame.pack_forget()
    self.register_frame.pack_forget()
    self.pacemaker_sel.pack_forget()
    self.mode_sel.pack_forget()
    self.display_frame.pack_forget()
    self.egram_frame.pack_forget()
    self.confirm_password_entry.delete(0, 'end')
    self.register_password_entry.delete(0, 'end')
    self.register_username_entry.delete(0, 'end')
    self.password_entry.delete(0, 'end')
    self.username_entry.delete(0, 'end')
    # self.pacemaker_interface.pacemaker_entry.delete(0, 'end')
    target_frame.pack()
```

Figure 79: Commented out line that had entry clearing upon leaving pacemaker selection page

Test 4 (Passes):

Submission of Different Pacemaker: Testing was done by typing in the entry box a different pacemaker name then the previously pacemaker name.

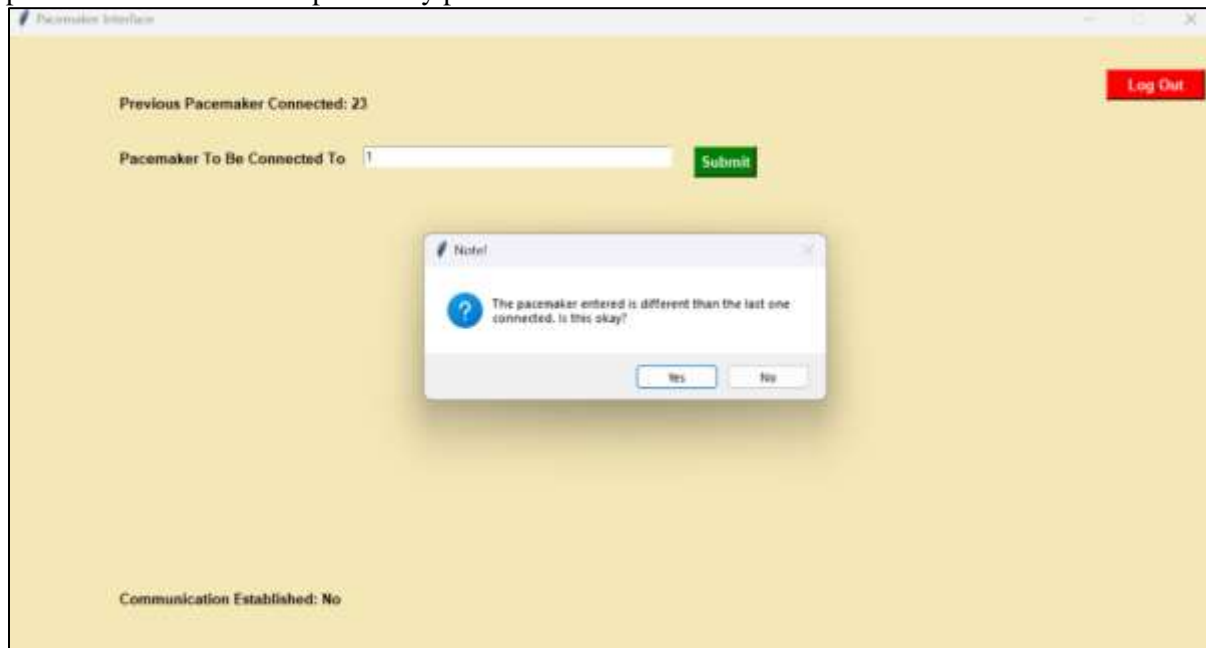


Figure 80: Warning about difference pacemaker connection

If the current pacemaker is different from the previous pacemaker, the code would throw in a message saying that the pacemaker is different from the old one.

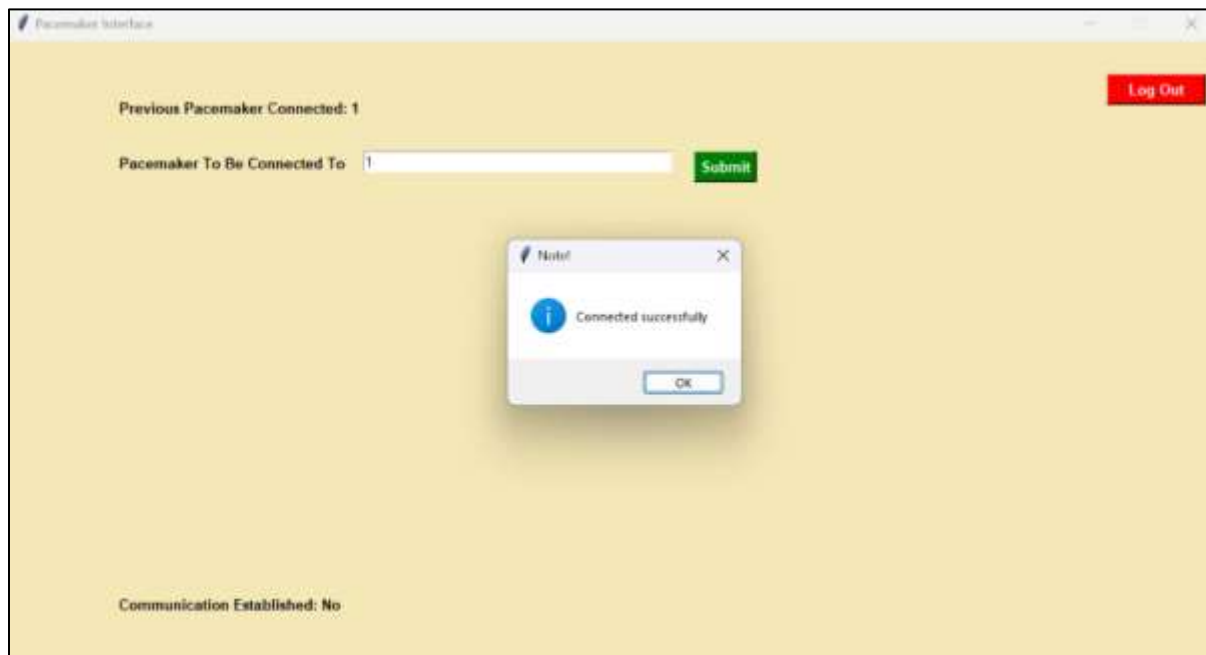


Figure 81: Successful Connection Message

If yes is clicked, then another message will be thrown, and it will direct the user to the selection mode interface.

Test 5 (Passes):

Submission of the same pacemaker: Testing was done by typing in the entry box the same pacemaker name then the previously pacemaker name.

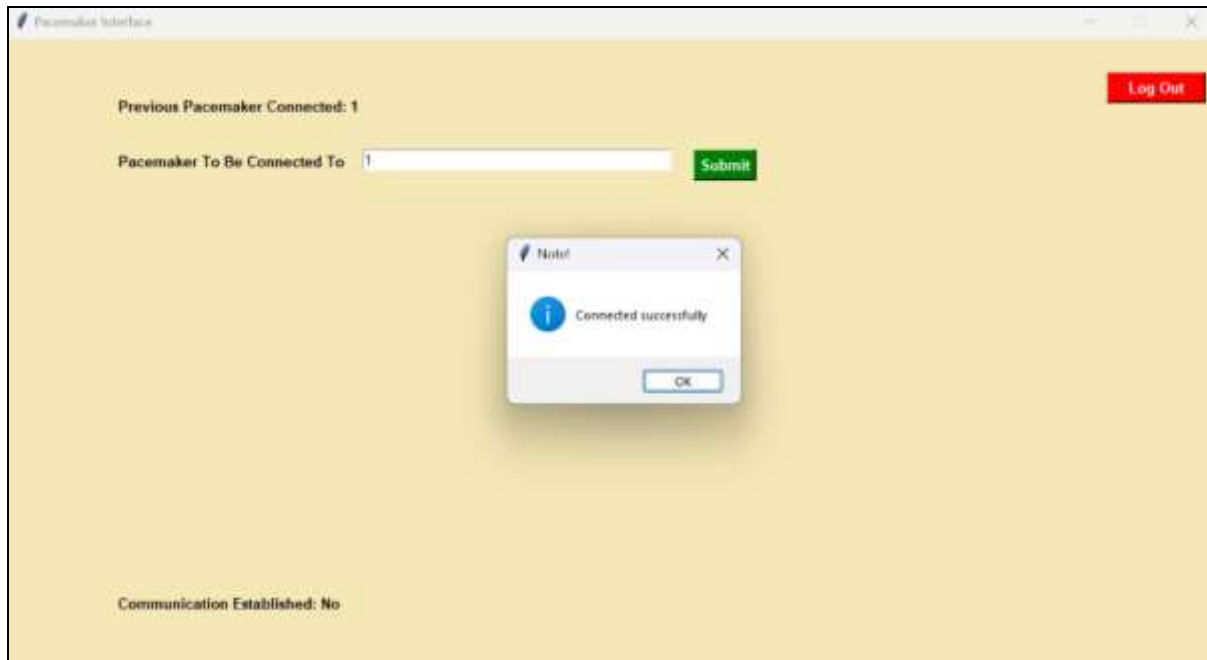


Figure 82: Successful Submission of Same Pacemaker

If the current pacemaker is the same as the previous pacemaker, the code would throw in a message saying that the user has been connected successfully and it moves onto the mode selection interface.

Test 6 (Passes):

Checking Pacemaker Connection: Testing was done by connecting and disconnecting the pacemaker board physically and checking whether the connection label changes in the right manner.



Figure 83: Pacemaker Interface when Pacemaker Connected

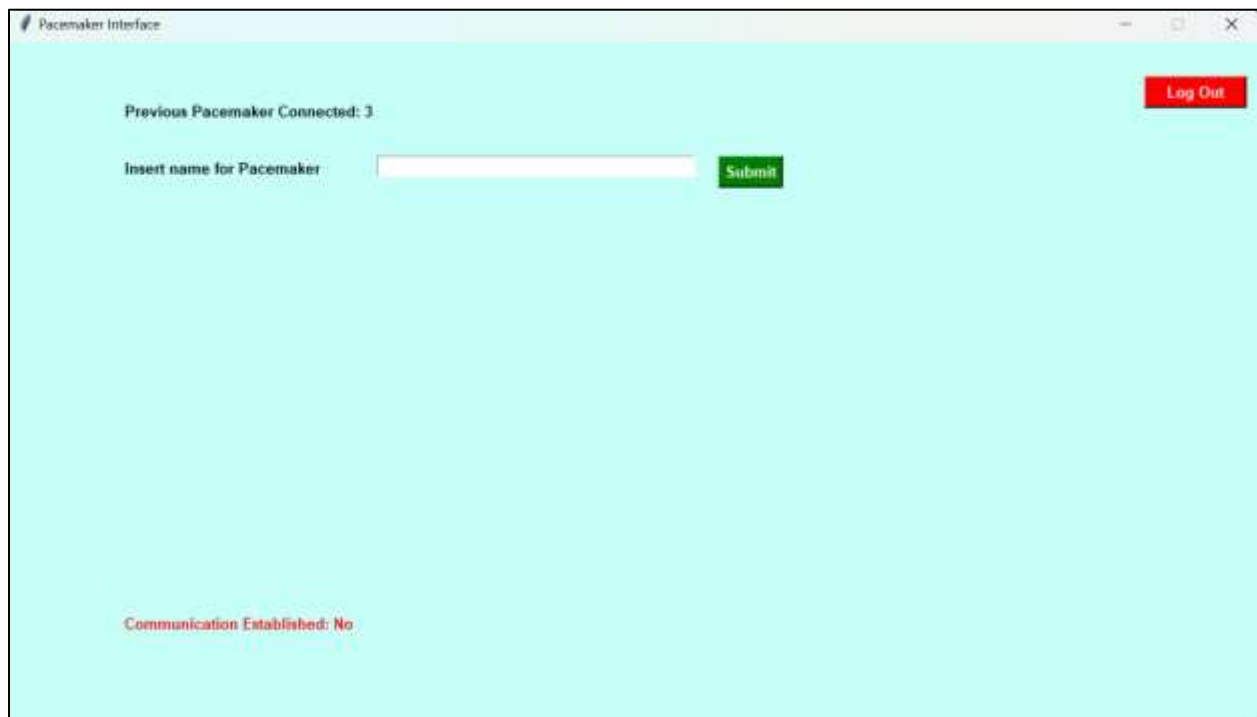


Figure 84: Pacemaker Interface when Pacemaker Disconnected

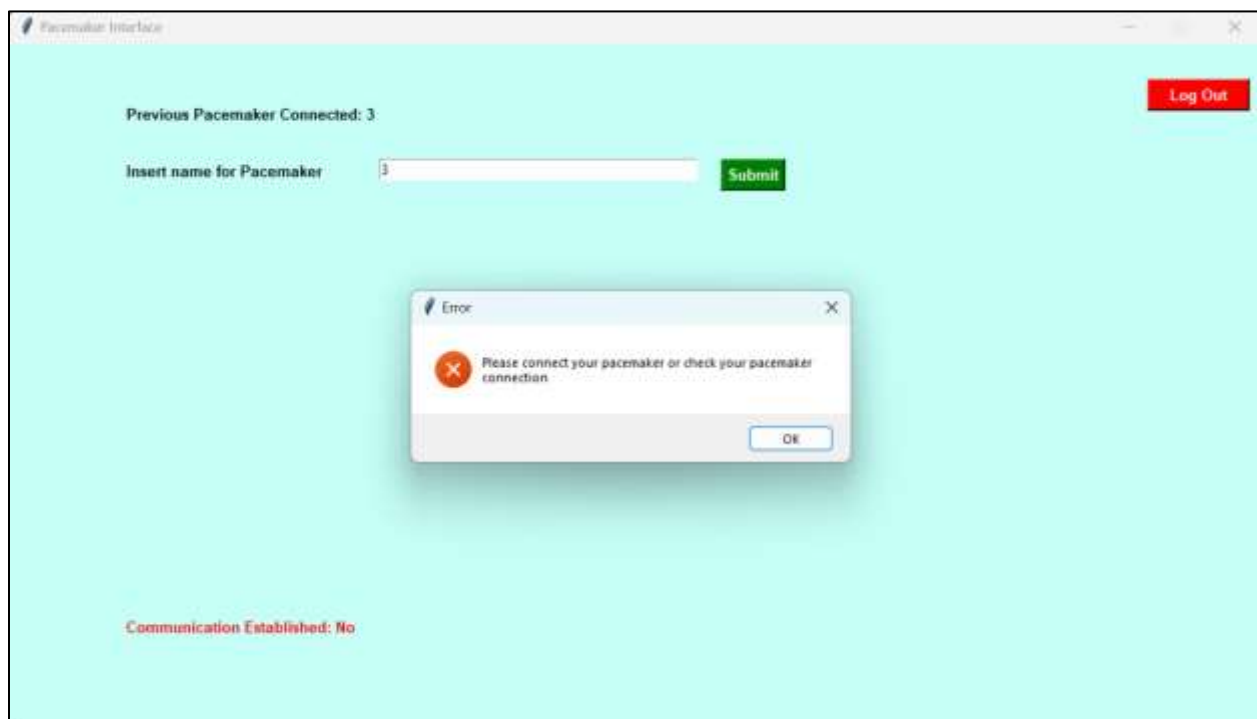


Figure 85: Warning showing the user cannot proceed further without connecting Pacemaker

The software shows to change the connection label very responsively and throws an error if the user tries to continue without connecting a board.

Test 6 (Failed):

Checking whether the DCM recognizes new devices: Testing was done by connecting new pacemaker and heart boards in order to see whether it would throw warnings.

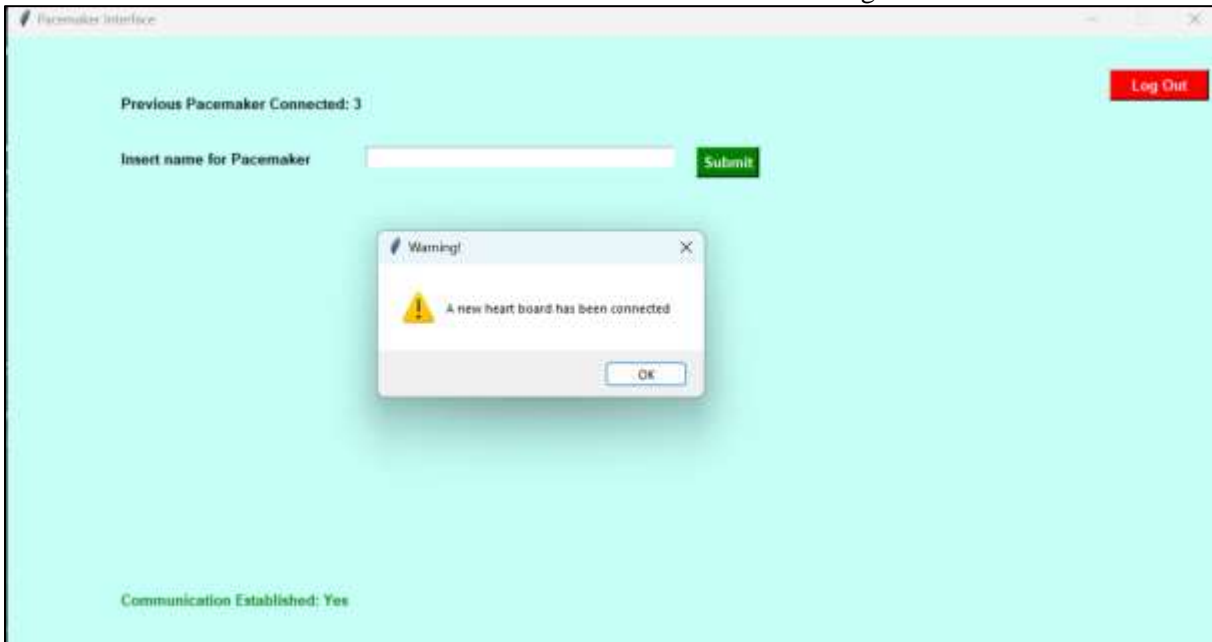


Figure 86: New Heart Board Warning

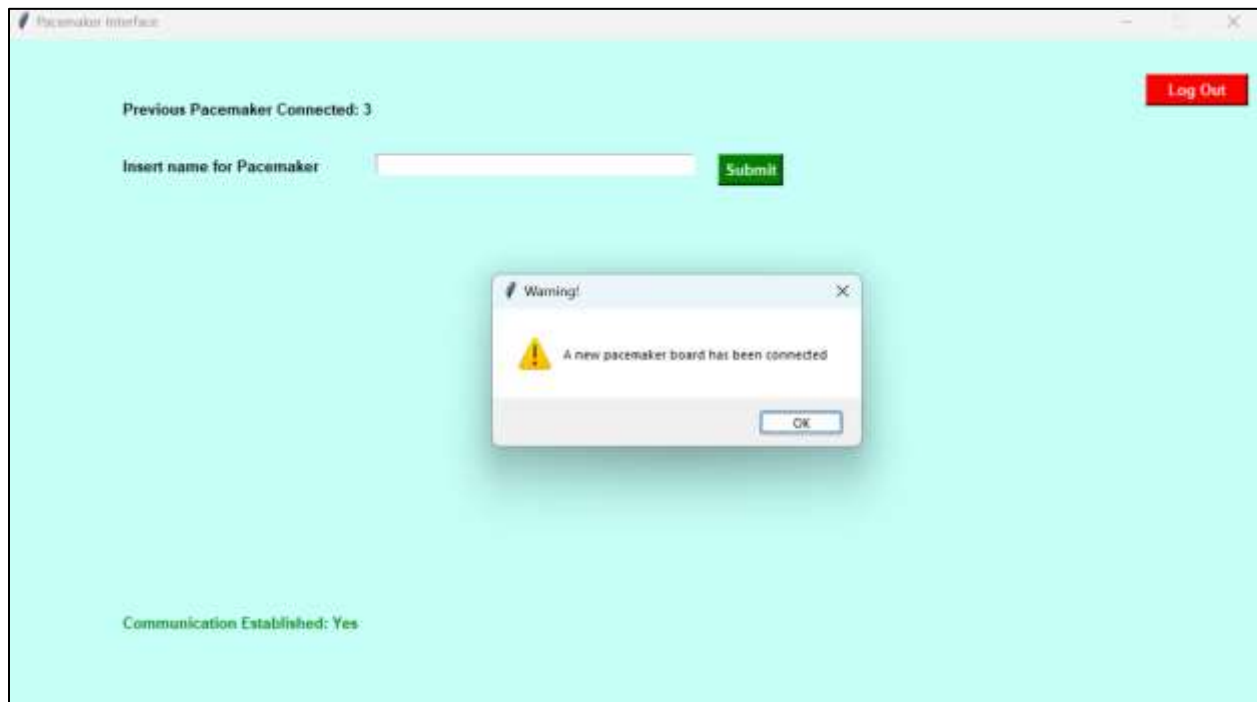


Figure 87: New Pacemaker Board Warning

When the database is fully cleared these two warnings show up, however, after connecting other boards, this warning fails to show. This is due to the fact that all boards used by us, and our colleagues happen to have the same hardware IDs which is the characteristic that this software distinguishes the devices by. This was later confirmed through print statements. Therefore, no software fixes were needed.

main.py:

Purpose:

The purpose of this module is to create and manage the main GUI application for the Pacemaker Interface. It basically serves as the entry point for the application, and it is responsible for handling the user login, registration, and routing between the different frames within this application.

Public Functions:

def __init__(self): This function serves as the constructor, and it is executed when an instance of the class is created. It is responsible for initializing the main application windows and setting up various GUI elements and frames. `super().__init__()` calls the constructor of the parent class to initialize the main application window.

def login(self): This function is responsible for processing the user's login credentials entered into the GUI which is the username and password. It also retrieves the username and password entered by the user and it passes it to the "login_user" function of the "userManager" class to create a new user account.

def register(self): This function handles user registration. It retrieves the new username, password and the confirmation password which is entered by the user, and it then calls the "register_user" function of the "userManager" class to create a new user account.

def route(self, target_frame): This function allows for the navigation between the different frames of the application. It hides the current frame and shows the target frame which is specified as an argument.

Global Variables:

self.login_frame: Represents a frame for the login interface

self.register_frame: Represents a frame for the registration interface

self.pacemaker_sel: Represents a frame for pacemaker selection

self.mode_sel: Represents a frame for mode selection

self.display_frame: Represents a frame for displaying data

self.egram_frame: Represents a frame for displaying electrogram data

self.username_entry: A text entry widget for entering the username

self.password_entry: A text entry widget for entering the password

self.register_username_entry: A text entry widget for entering a new username

self.register_password_entry: A text entry widget for entering a new password

self.confirm_password_entry: A text entry widget for confirming the new password

self.login_button: A button widget for initiating the login process

self.register_button: A button widget for initiating the registration process

self.register_button2: A button widget for the completion of the registration process

self.back_button: A button widget for navigating back to the login interface from the register interface

self.logo_image: An image widget for displaying the McMaster University logo

Private Functions:

There are no private functions in this module.

Testing and Results:

The testing done in this module involves verifying the functionality of login and registration processes and ensuring that the frame navigation works as expected. Cases for successful login, failed login, successful registration, and other edge cases were tested. These tests determined whether the application was working correctly or not. This module also runs the entire application as well as it incorporates all the different modules into this module.

Test 1 (Passes):

Registering more than 10 users: The maximum number of users that can be registered is 10, so we tried to register more than 10 users.



Figure 88: Error of Maximum Users Reached

Using an if statement, we were able to throw in an error message when 10 users were reached and it did not allow the user to save more users.

Test 2 (Passes):

Registering a different account with a username that already existed within the JSON file: Testing was done by registering the same username which had already been registered before again.

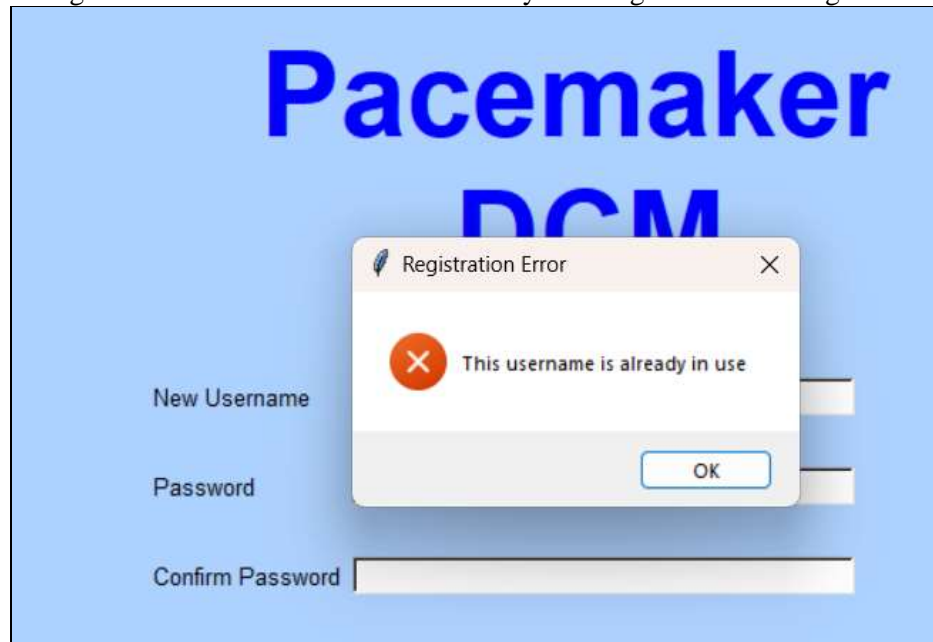


Figure 89: Username already in data storage error

At first, our code allowed same usernames to be registered, but by throwing in an if statement, we were able to allow only one user of the same username to be registered and if the user tried registering with the same username, an error message would be displayed.

Test 3 (Passes):

Entering different and same password: Testing was done by entering different passwords for the “Password” and “Confirm Password” entries and entering the same passwords for the “Password” and “Confirm Password” entries.



Figure 90: Error Message of Different Password Submissions During Registration

If different passwords were entered, an error message would be thrown saying the passwords don't match, but if the same passwords were entered, then the user would be created, and a message would be displayed saying that the user was created successfully.

Test 4 (Passes):

Invalid/Wrong Login Credentials: Testing was done by entering the wrong username and passwords.

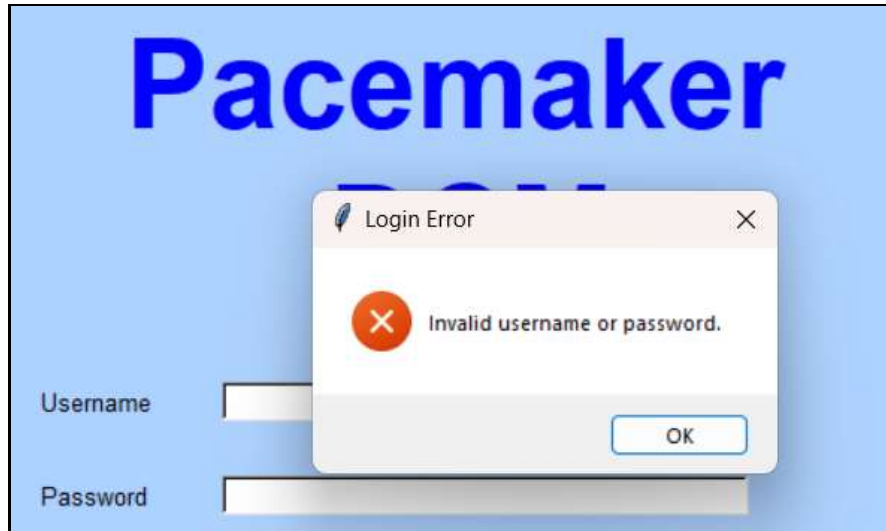


Figure 91: Error Message when Non Existing User Info is Entered

If the wrong username and password were entered, the code would throw in an error message saying that the username and password given are invalid, but if the username and password are right, then it would throw in a message saying that you have successfully logged in.

Test 4 (Failed):

Username and password registration: Testing was done by clearing the user_data JSON file.

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.10_3.10.3056.0_x64__qbz5n2kfra8p0\lib\tkinter\__init__.py", line 1921, in __call__
    return self.func(*args)
  File "c:\Users\livir\OneDrive\Desktop\Pacemaker\DCM\main.py", line 109, in register
    user_manager.register_user(username,password,confirm)
  File "c:\Users\livir\OneDrive\Desktop\Pacemaker\DCM\user_manager.py", line 35, in register_user
    self.users.append({"username": username, "password": password})
AttributeError: 'dict' object has no attribute 'append'
```

Figure 92: Error thrown when registering with a clear user_data file

We later realized that this error was not an error within the code. Instead, it is an error introduced from how we leave the empty JSON file. We found that if the JSON file is initiated as a dictionary with {}, this error will be thrown. However, if the JSON file is initiated as a list with [], the software acts as intended. So no software fixes were needed. This was a test that we failed during our demo however it has been documented to show it has been fixed.

user_manager.py:

Purpose:

The purpose of this module is for managing the user registration and login in the application. It provides then functions to register new users, verify login credentials, load, and save user data.

Public Functions:

def __init__(self, user_data_file, main_app): This constructor initializes the user manager which takes two parameters: user_data_file and main_app. The user_data_file is the path to the user data file and main_app is a reference to the main application.

def register_user(self, username, password, confirm): This function registers a new user which takes three parameters: username, password and confirm. This function checks if the maximum number of users is reached which is then and if it is, then it shows an error message saying that the maximum number of users is reached. It also verifies the provided username is unique and if not, then it shows an error message. Also validates if both the username and password are provided. Also checks if the password and the confirmation match. If all the conditions are met, it adds the user to the user list and saves the data, and at the end it shows a success message.

def login_user(self, username, password): This function verifies user login credentials which takes 2 parameters: username and password. This function iterates through the list of registered users and checks if the provided username and password match any of the existing/registered users. If a match is found, it routes to the pacemaker selection frame and shows a success match, if not match is found, then it shows an error message.

def load_user_data(self): This function is responsible for loading user data from the JSON file. It attempts to open the user data file specified during the initialization of the user manager and if the file is found, it then reads the data and the list of users is loaded into the self.users variable, but if the file is not found, then an error is thrown and an empty list is returned.

def save_user_data(self): This function is used to save the user data back to the JSON file. It opens the user data file in write mode and it pushes the self.users list into JSON format and it writes it to the file. This function is called when new users are registered to ensure that the user data file is updated with the latest/newest information.

Global Variables:

self.user_data_file: This global variable holds the path to the user data file. It is also used by both the load_user_data and save_user_data functions to read and write the user data.

self.users: This global variable stores the list of registered users and it is initially loaded from the user data file using the load_user_data function and it is updated when the new users are registered using the register_user function.

Private Functions:

There are no private functions in this module.

Testing and Results:

In this module, testing would be done on the user interactions with the registration and login processes, and messages are displayed accordingly such as the registration success, then error due to duplicate usernames, or login success/error messages. Testing was done to see if an error message is thrown if more than 10 users are created, tests are done on checking if a valid username is given, if not it should throw an error message, then if the password is valid and the password and confirmation password both match each other. The testing of this module goes hand in hand with the test cases mentioned in the main.py module.

display.py:

Purpose:

The purpose of this module is creating a GUI for displaying parameters of an existing pacemaker. This module allows users to select a pacemaker from a dropdown menu and then click a button to display the parameters that are associated with the selected pacemaker. This module is for managing and interacting with the pacemaker data, but not editing any of that pacemaker data.

Public Functions:

def __init__(self, root, main_app): This constructor function initializes the display module and sets up the GUI components including dropdown menu, buttons, labels with the given “root” frame and the main application object called “main_app”. This function sets the selected pacemaker variable, creates buttons and labels for display, and it also loads the list of existing pacemakers from a JSON file during the initialization.

def load_existing_pacemakers(self): This function takes only one parameter “self”, and this function loads the existing pacemakers’ names from a JSON file and returns them as a list. It attempts to read a JSON file which is named “pacemaker_data.json” and it retrieves the names of existing parameters. If the file exists, the names are returned as a list and the list will be used to populate the dropdown menu in the application. It opens and reads the JSON file, and the function returns the list of the pacemaker names and if it is empty, it returns an empty list.

def display_parameters(self): This function is called when the Display button is clicked. This function retrieves the selected pacemaker, and it displays its parameters in a label. This function fetches the selected pacemaker, it reads its parameters from a JSON file, and it displays the parameters for the selected pacemaker in a label.

Global Variables:

self.selected_pacemaker: This global variable is a tkinter variable that holds the selected pacemaker from the dropdown menu. It provides a way to access the selected value and it is used to update the selection as the user interacts with the dropdown menu.

self.label3: This global variable is a tkinter label used to display the parameters. This is for the parameters that will be shown when the Display button is clicked.

self.existing_pacemakers: This global variable is a list containing the names of existing pacemakers which are loaded from a JSON file. This is populated by load_existing_pacemakers function.

Private Functions:

This module does not contain any private functions.

Testing and Results:

For this module, testing was done on verifying that the GUI components are created correctly and checking if clicking that “display” button shows the parameters for the selected pacemaker. The application was run, and the user interacted with the GUI to make sure that it behaved as expected. Checking if clicking the “display” button leads to another frame, then checking if the previous pacemakers were options in the dropdown menu and then when one of the pacemakers was clicked, it accurately displays the updated parameters of that specific pacemaker.

Test 1 (Passes):

Enter data and check if data saved by trying to display it: Testing was done by submitting data for one run and then terminating the application and checking to see if the previous runs data can be displayed. This test had passed when we had run the application. This was expected as the function purely was based on the currently working files. In addition, you may realize that if data entered on current run, it won't show until you restart. This was a design decision that we had made to show only ‘past inputs’ stored.

Test 2 (Passes):

Enter an invalid pacemaker and see if it appears in the dropdown option for displaying pacemakers: We had tested this by entering an invalid pacemaker name value into the entry box, which at first gave us a pop error which is intended. Then later we had checked to see whether the invalid pacemaker had been entered into the file. In this case, it did not submit into the file which means the test passed.

Test 3 (Failed - Required Fix):

Clear all data in JSON and check how it handles empty JSONs: This test was done by emptying out the pacemaker_data.json file and then entering the display screen to see what would happen. At first, it gave us the error shown below if the JSON didn't exist or was empty.



```
File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.10.3.10.3056.0_x64_gbz5n2kfratp0\lib\json\__init__.py", line 346, in loads
    return default_decoder.decode(s)
File "C:\Program Files\WindowsApps\PythonSoftwareFoundation.Python.3.10.3.10.3056.0_x64_gbz5n2kfratp0\lib\json\decoder.py", line 337, in decode
    obj, end = self.raw_decode(s, idx=_w(s, 0).end())
```

Figure 93: JSON Decoding Error

This was due to the string variable being used in the dropdown returning a ‘NoneType’ which the JSON decoder didn’t accept. To fix this, we added a conditional shown below:


```
# Create a dropdown menu to select an existing pacemaker
self.selected_pacemaker = tk.StringVar(root)
if self.existing_pacemakers == []:
    self.label1 = tk.Label(root, text="No Existing Data Found", font=font)
    self.label1.place(x=400, y=250)
    self.log_out = tk.Button(root, width='10', border=2, text="Log Out", command=self.logout)
    self.log_out.place(x=1000, y=30)
    self.back_button = tk.Button(root, text="Back", width='10', border=2, command=self.back)
    self.back_button.place(x=900, y=30)
    return
else:
```

Figure 94: Conditional that checks if no pacemakers exist in data

After adding that fix to the logic, it would display the following if there was no data:



Figure 95: Error message shown when no pacemaker exists

This logic would check to see whether the list extracted from the file is an empty list, where if the conditional is true, it would display the message above.

Test 4 (Failed - Required Fix):

Test how data for modes like VVI is positioned on the screen (does it wrap around or get cut off)?: This was tested by submitting data from all modes and seeing how it would display on the screen. At first data was getting cut off but after setting a wrap length on the label as shown below, the issue was resolved:

```
self.label3.config(wraplength=1000)
```

Figure 96: Command to make label wrap when 1000pixels of width covered

This allowed the label to get cutoff and transfer onto the next line at certain amount of length. With these test cases, we were able to verify that our display.py module works as intended and handles common errors that could occur.

Test 5 (Pass):

Testing to see user specific data display: This was tested by submitting multiple sets of data for each user through multiple users and checking whether the displayed data corresponds with what was submitted for the respective user.



The screenshot shows the 'Pacemaker Interface' window. At the top, there are two green buttons: 'View Egram Data' and 'Display Existing Data'. To the right are 'Back' and 'Log Out' buttons. Below the title bar, there is a 'Select Mode' section with a blue 'ADD' button and a blue 'Next' button. In the center, there are four input fields for parameters: 'Lower Rate Limit' (50), 'Upper Rate Limit' (120), 'Atrial Amplitude' (5.0), and 'Atrial Pulse Width' (1). Below these fields is a 'Submit Parameters' button.

Figure 97: Data Submission



The screenshot shows the 'Pacemaker Interface' window. At the top, there is a message: 'Please Choose a Pacemaker in order to view past data'. Below this message are two buttons: a blue '3' button and a green 'Display' button. To the right are 'Back' and 'Log Out' buttons. Below the buttons, there is a section titled 'Your data will appear below :'. Inside this section, there is a table displaying the submitted parameters:

ADD	
Lower Rate Limit	50
Upper Rate Limit	120
Atrial Amplitude	5.0
Atrial Pulse Width	1

Figure 98: Data displayed

We can see that only the data submitted by the user on pacemaker 3 is visible on the screen when submitted.

Test 5 (Pass):

Testing to see whether the display screen is scrollable: This was tested by submitting data for all modes for one user and seeing whether the user would be able to scroll the display screen to display all the data.



Figure 99: Visible data but showing excess towards the right

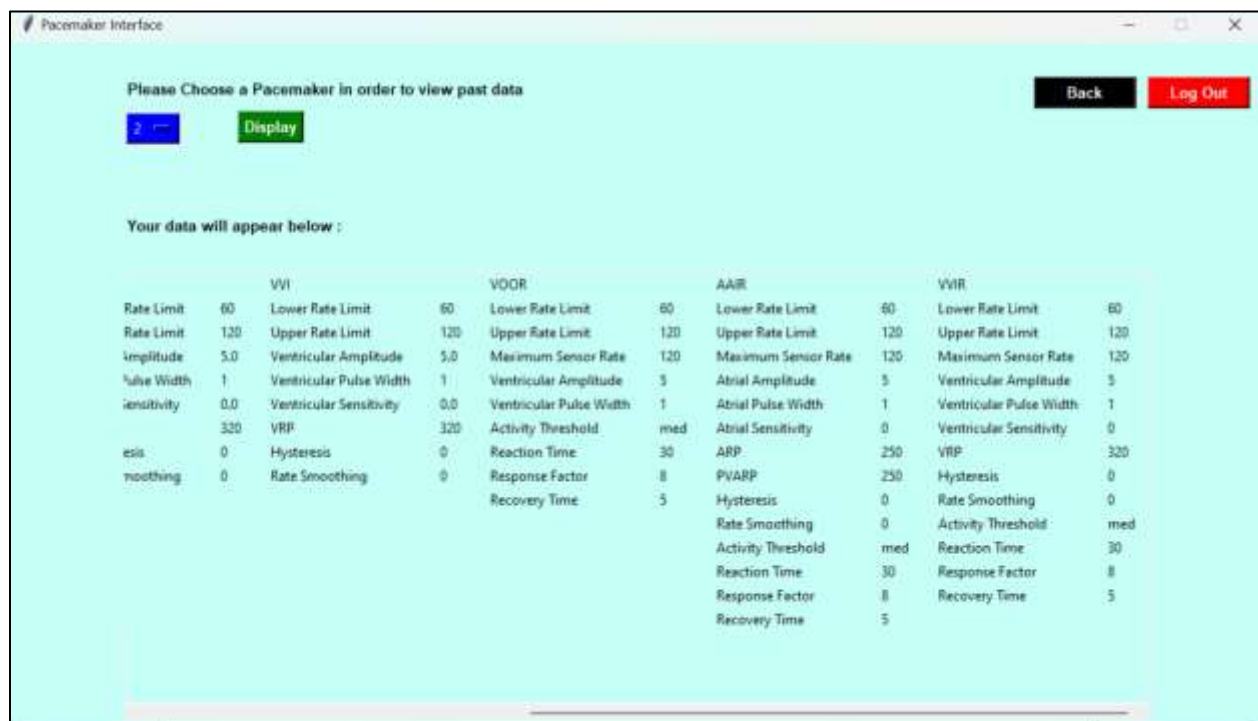


Figure 100: Showing the ability to scroll showing more submitted data

egram.py:

Purpose:

The purpose of this module is to provide a GUI for displaying an electrogram (EGM) or a graph of voltage over time. It is responsible for visualizing and presenting the electrical signal recording generated by a pacemaker over time. This allows users to view and interact with the EGM data in a user-friendly manner. The updated module significantly enhances its functionality by offering a more detailed and dynamic representation of the electrogram data. It is designed to provide healthcare professionals with a comprehensive view of a patient's cardiac electrical activity, recorded by a pacemaker. This visualization is crucial for diagnostic and monitoring purposes.

Public Functions:

def __init__(self, root, main_app): This constructor function takes two parameters: “root” and “main_app”, which represents the root frame of the GUI and the main application. It initializes the Egram module within the application. It creates a Matplotlib figure and axes, sets labels for the graph’s x and y axes, generates a canvas for rendering the figure, and places the canvas in the center of the root frame for display. It also creates a log-out and back button for navigation for the application. The state variables are used to set up and display the graph and interact with the root frame and main application. The constructor has been expanded to include the initialization of a new SerialCommunication instance, reflecting the module's capability to communicate with external devices, which in this case is a pacemaker. It sets up a more elaborate graphical representation with two subplots for atrial and ventricular data, which provides a more detailed analysis. The introduction of a dropdown menu for mode selection (Atrial, Ventricular, Atrial + Ventricular) adds versatility to the visualization, providing specific data requirements. A new button for starting/stopping the graph provides better control over the data visualization process.

def render(self, mode): This function now adapts the graphical display based on the selected mode, showing either atrial, ventricular, or both data sets. It includes logic to stop the graph when switching modes, ensuring a seamless transition. The implementation of dynamic positioning and visibility of the graphs based on the mode enhances the user experience.

def update_egram_data(self, mode, start_time): This function illustrates the module's capability to fetch real-time Egram data from the pacemaker. It dynamically updates the graph based on the selected mode, showcasing the module's ability to handle live data effectively.

def update_subplot(self, axis, new_data, plot_title): This function is responsible for updating the individual subplots with new data. It shows the module's capacity to process and visualize real-time data efficiently.

def toggle_egram(self): A new function that provides the ability to start or stop the graph, enhancing user control over the data visualization process.

def stop_egram(self): This function allows for the stopping of the graph, ensuring that resources are not unnecessarily used when the graph is not needed. This also allows the system to work faster and smoother seamlessly.

def log_out_command(self) and def back_button_command(self): These functions enhance the user navigation experience by ensuring the graph is stopped when navigating away, preventing potential

issues with continued data fetching or visualization when the graph is not in view, which can result in the slowness of the system.

Global Variables:

self.root: This global variable stores a reference to the root frame of the GUI. It is used to fix the Egram module within the entire application.

self.main: This global variable holds a reference to the main application instance, and it is used for navigating between the different frames or components of the application.

self.fig: This global variable is an instance of a Matplotlib Figure which represents a graphical figure which will be used to plot the EGM data

self.ax: This global variable is an instance of a Matplotlib Axes which is used for setting the labels for the x and y axes of the graph

self.canvas: This global variable is a Matplotlib canvas created to display the Figure, and it also allows for the rendering and display of the EGM graph within the tkinter user interface.

self.canvas_widget: This global variable represents the tkinter widget created from the Matplotlib canvas, and it is positioned in the center of the root frame for displaying the EGM graph.

self.log_out: This global variable is a button widget for logging out of the application. It triggers the self.main.route method to navigate back to the initial/login frame when the logout button is clicked.

self.back_button: This global variable is a button widget which allows users to navigate back to the mode selection frame, and this also uses the self.main.route method to switch between the frames.

self.serial_comm: A new global variable that establishes a communication link with the pacemaker, indicating the module's expanded capability to interact with external hardware.

self.ax_atrial, self.ax_ventricular: These variables have been modified to support two subplots, reflecting the module's enhanced capability to display more complex data.

self.mode_label, self.mode_var, self.modes, self.mode_dropdown, self.next_button: These variables are part of the newly introduced dropdown menu for mode selection, enhancing the module's interactivity and functionality.

self.lines_atrial, self.lines_ventricular: Variables to manage the data lines in the subplots, ensuring dynamic data visualization.

self.timer_interval, self.egram_data, self.stop_graph, self.stop_button: New variables introduced to manage the real-time data fetching and updating mechanism, demonstrating the module's improved functionality.

Private Functions:

This module does not contain any private functions.

Testing and Results:

For this module, testing was done on ensuring that the graphical display correctly visualizes the EGM data which for now is just a placeholder graph. Testing includes checking the appearance and interactivity of the graph. Testing was done on confirming that the buttons for logging out and returning back to the Mode Selection frame are working as expected.

Test 1 (Passes):

As this module exists as a placeholder for future data, the only test case we had is how the graph looked on the frame and if the correct axis and scale was being displayed which it passed.

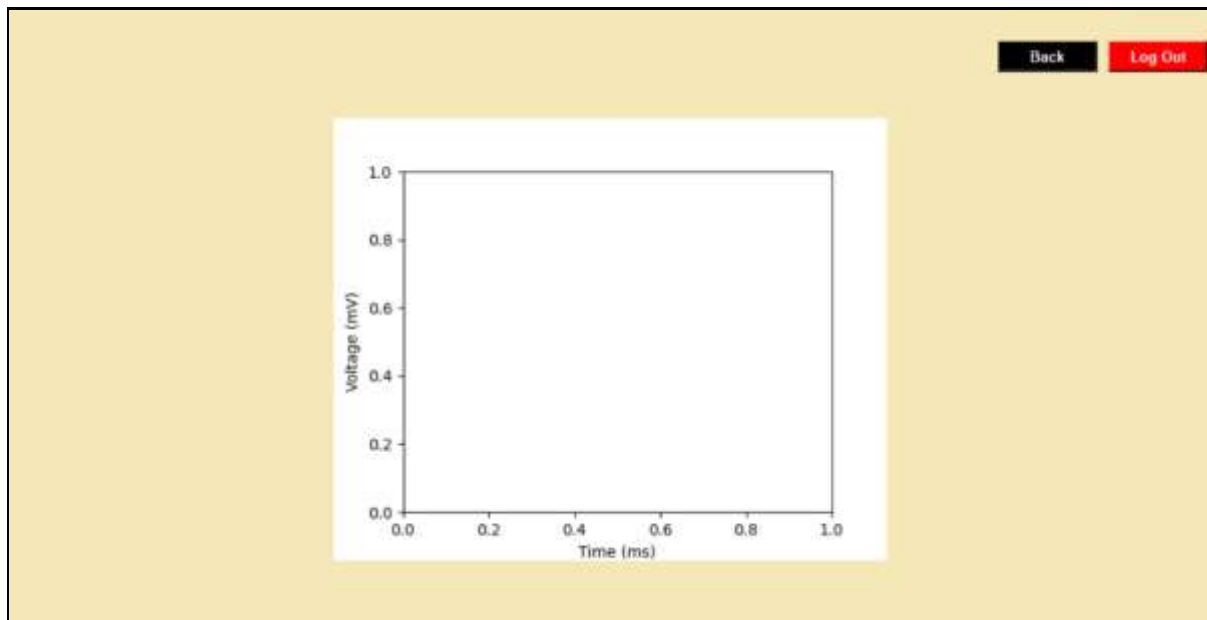


Figure 101: Placeholder Graph for Electrogram Data

Test 2 (Passes):

Check if Egram Graph is Looking like Pulses when Connected with Heartview: To get clean looking pulses, we receive data from the A0 and A1 pins from the heart when Heartview is connected to the DCM and we send DCM data to the pacemaker. The image below shows the result of such a test:

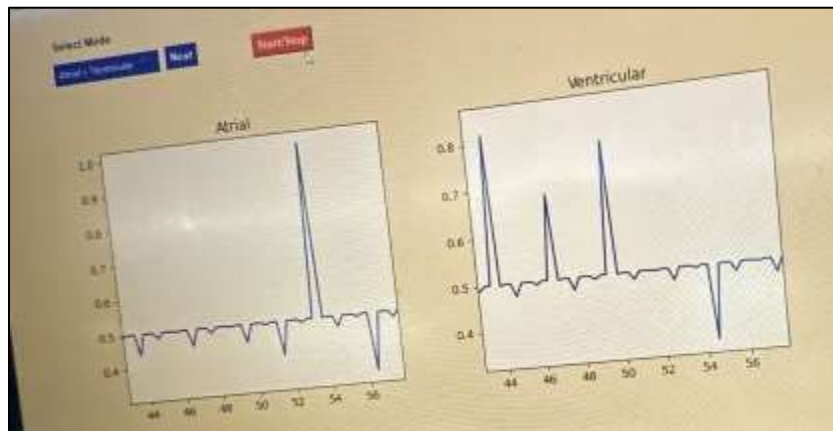


Figure 102: Correct Egram Pulses on Graph for VOO Mode

Test 3 (Passes):

Disconnect Pacemaker while egram data is transmitting data for the graph: Transmission should stop and there should be an error message that shows up to warn the user to connect the pacemaker back.

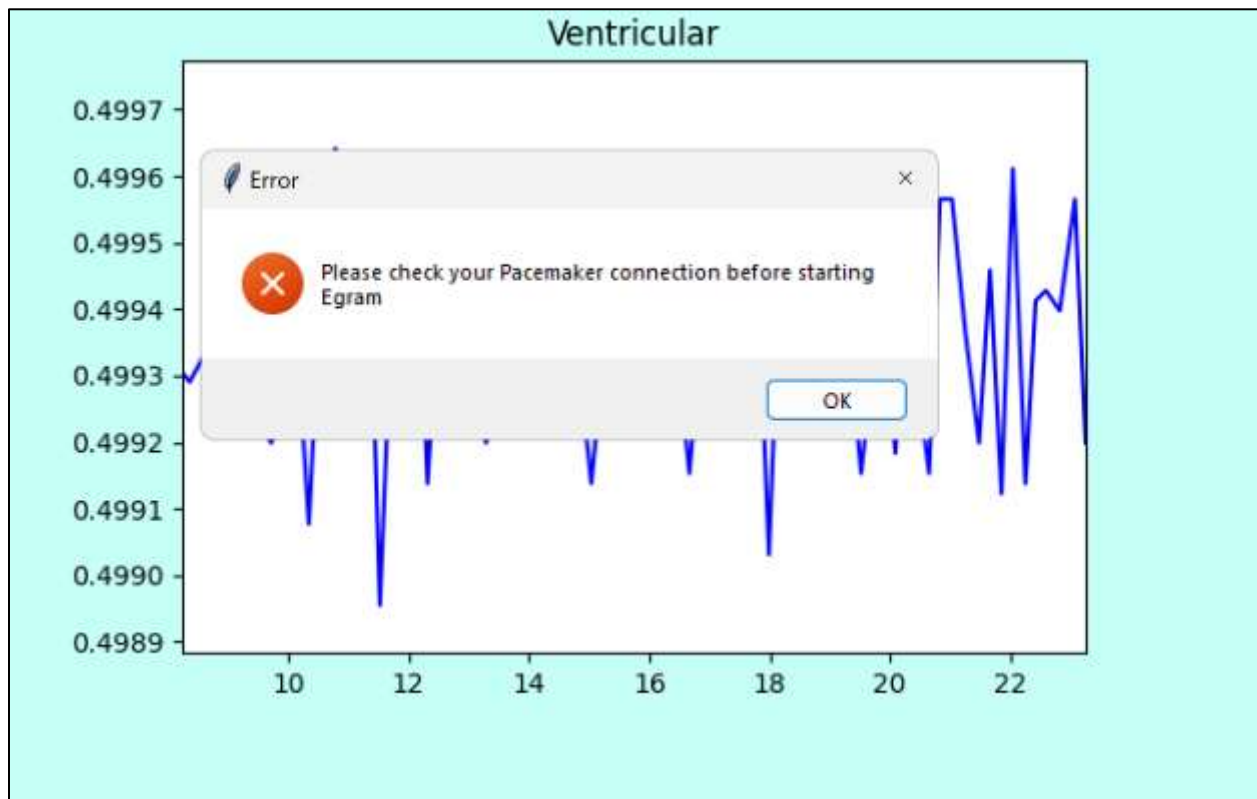


Figure 103: Check Pacemaker Connection Error Message

Test 4 (Passes):

Egram graph should stop when back or log out is pressed: We have a blue LED blink on the board when egram data is being transmitted. To test this feature, we pressed back and logout and saw that the blue LED stopped blinking meaning egram data successfully stopped transmitting.

Test 5 (Passes):

Tested Transmission between all Modes and the Dropdown Menu: We have a dropdown menu that lets you select which graph you want to display as shown below.

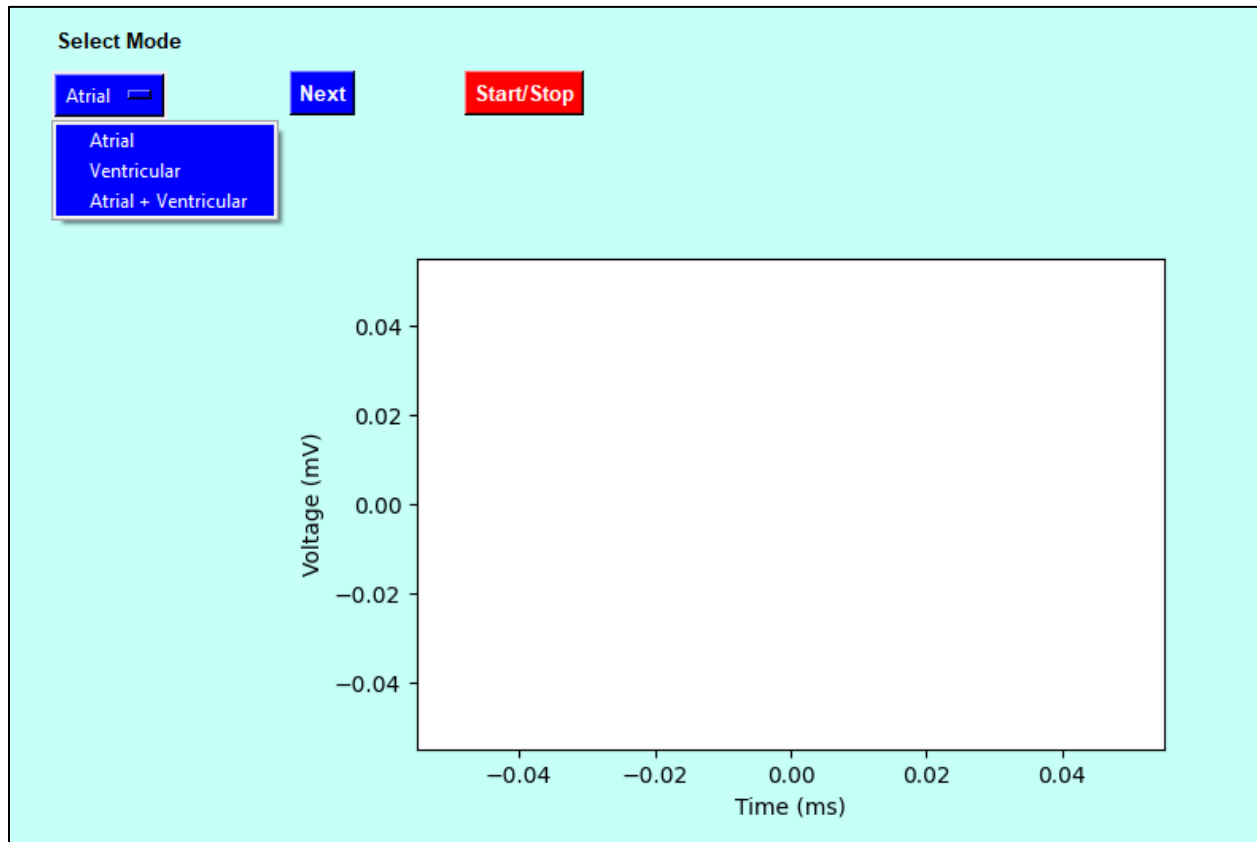


Figure 104: Graph Mode Dropdown List

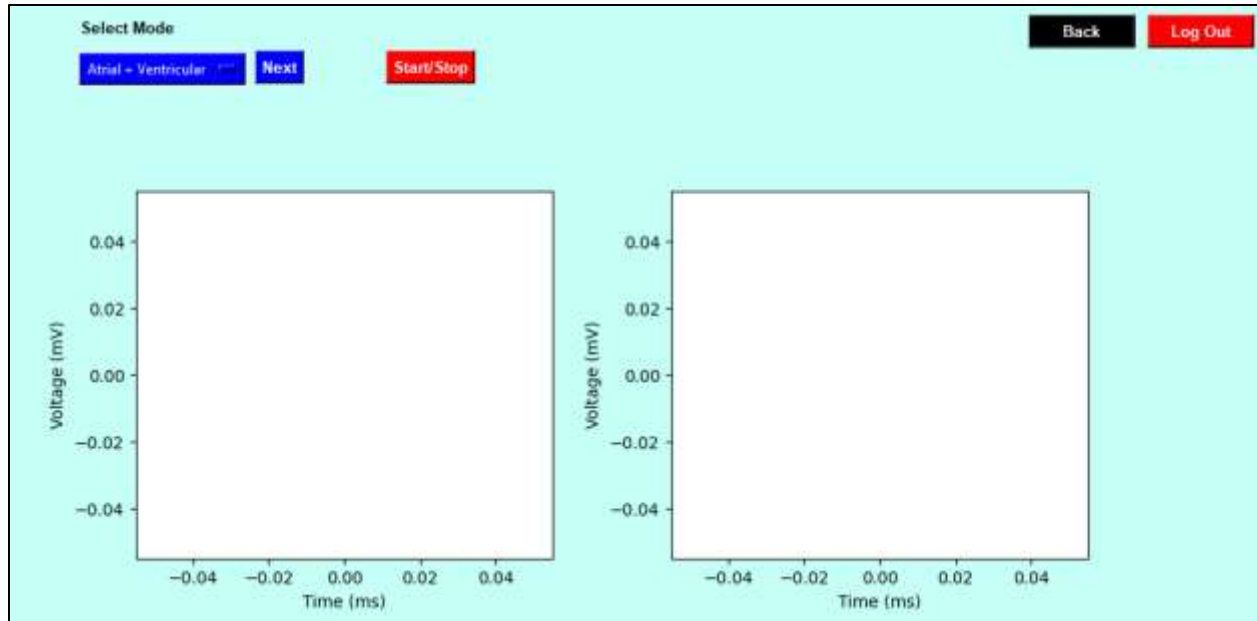


Figure 105: Two Graphs for Atrial + Ventricular

When we are in Atrial + Ventricular, we see two graphs. The one on the left transmits Atrial Data and the one on the right transmits Ventricular Data as seen in Figure 102.

Test 6 (Passes):

Tested the start/stop button for every mode: We have a start/stop button that will stop the dynamic graph at any point and can resume at any point. To make sure this was working as intended, we switched between all the modes and messed with this button and tried to get the program to behave incorrectly. We were unsuccessful in trying to break it.

`serial_communication.py`:

Purpose:

The `SerialCommunication` class in this module serves as an interface for managing serial communication between the main application and the external hardware devices, specifically the pacemaker in this case. It primarily focuses on detecting and establishing connections with these devices, handling user-specific configurations, and transmitting pacemaker parameters. The class also retrieves and validates data received from the pacemaker, ensuring accurate communication and effective monitoring of the pacemaker settings.

Public Functions:

def __init__(self, main_app): This function is a constructor for initializing the `SerialCommunication` class. It takes the `main_app` as a parameter, which is a reference to the main application, allowing this class to interact with other parts of the application through the main module.

def check_connection(self): This function scans for connected devices and it also verifies if a pacemaker is connected or not. It reads a JSON file to manage device connections for different users and it also shows warnings if new devices are connected. This helps in maintaining a record of the devices that is used by each and every user.

def send_parameters(self, data_to_send, s0, s1): This function handles the sending of pacemaker parameters to the device. It packages the data into a specific format and establishes a serial connection to transmit it. The function also reads back the parameters from the pacemaker to confirm that there was a successful transmission and it displays relevant messages to the user based on the success or failure of the operation.

Global Variables:

self.main: This global variable stores a reference to the main application. This allows the `SerialCommunication` class to interact with other components of the application and it also allows it to access the global data.

self.main.port_list: This global variable is an array to store the hardware identifiers of the connected devices. It is used to keep track of which devices are currently connected to the application.

self.main.pacemaker_port: This stores the serial port identifier for the different pacemakers. This variable is crucial for establishing a serial connection with the pacemaker and the different types of pacemakers.

self.main.heart_port: This global variable is similar to `pacemaker_port`, which stores the serial port identifier for the different pacemakers.

Private Functions:

This module does not contain any private functions.

Testing and Results:

Test 1 (Passes):

Disconnecting Pacemaker and Trying to Submit: If the user disconnects the pacemaker and tries to send programmable data to the pacemaker, our serial communication check_connection function should be able to display an error message saying that there is no pacemaker connected.

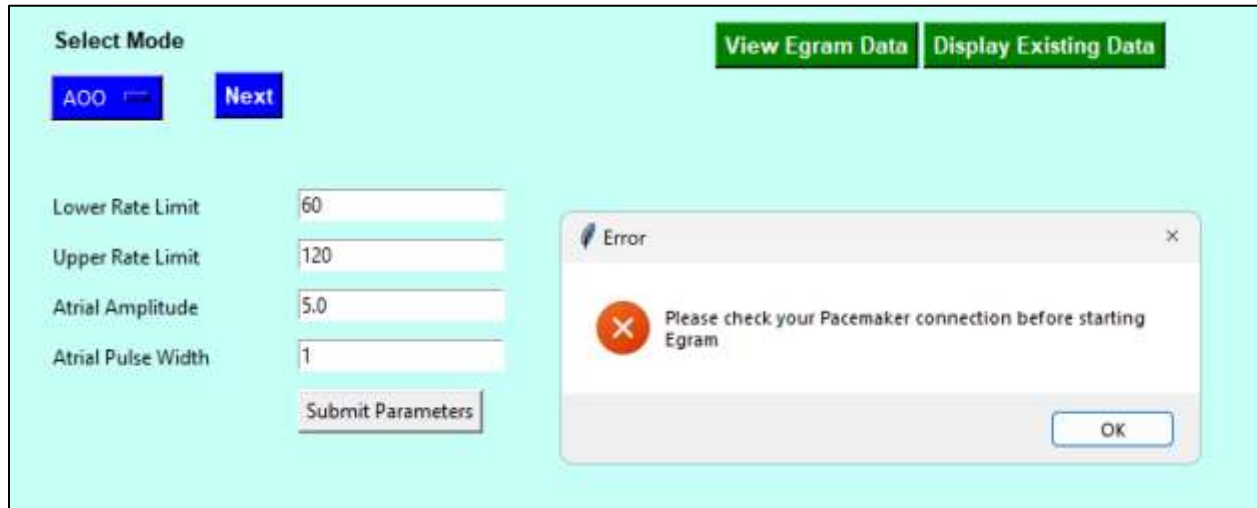


Figure 106: Disconnect Pacemaker before Pressing Submit

Test 2 (Passes):

Checking If Board Data Saved was User Specific: If the same board is connected with a new user logged in, it should still save the pacemaker as a new pacemaker.



Figure 107: Saves Pacemaker Data for New User

Test 3 (Passes):

Checking if data that was sent was received back successfully: Based on our assurance case diagram, we have conditionals in place to ensure we are receiving back data correctly. For this test, we tried to see if everything worked perfectly first by submitting parameters and looking at the message that popped up.



Figure 108: Successful Communication with Pacemaker

Then, we changed what we were receiving from Simulink to be incorrect data which then displayed the following error message:

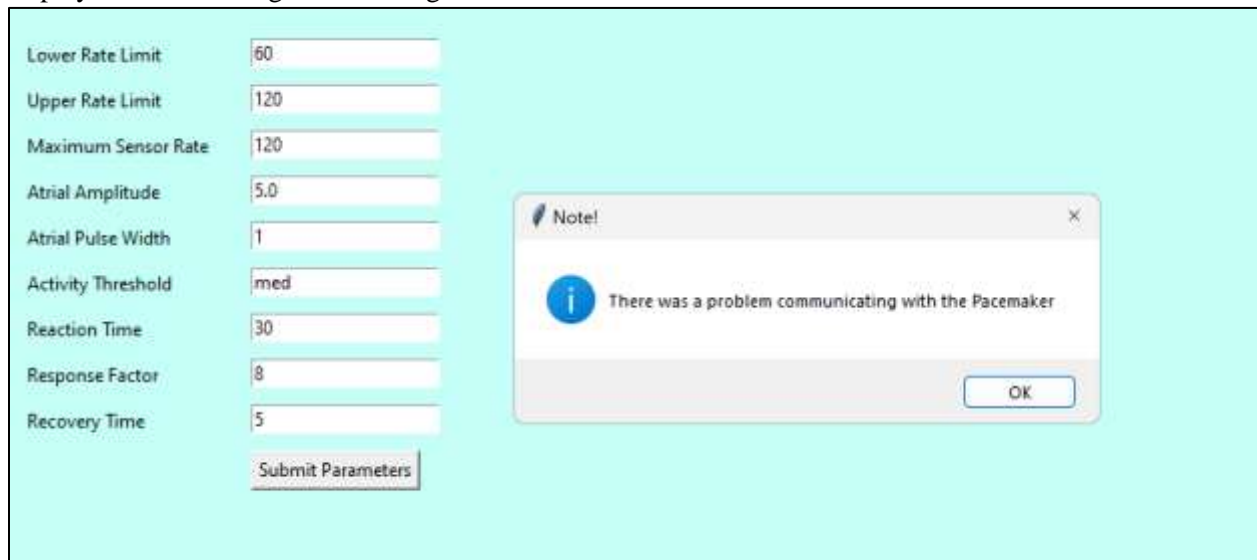


Figure 109: Unsuccessful Communication with Pacemaker

Appendix A: Simulink Sensing Circuit Model

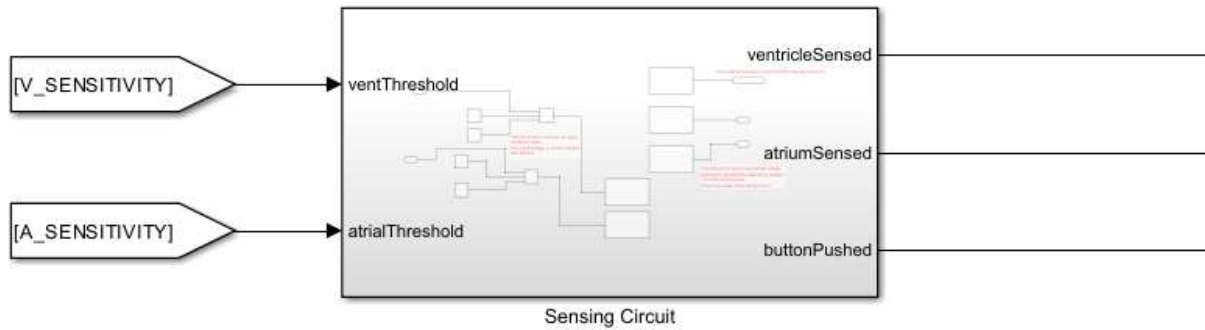


Figure 110: Sensing Circuit Exterior

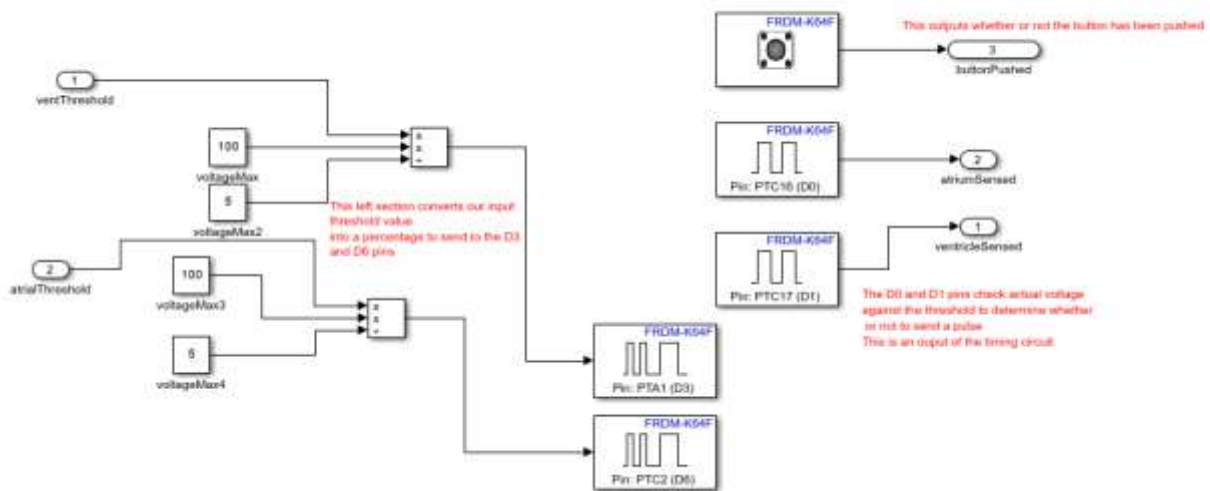


Figure 111: Sensing Circuit Interior

Appendix B: Simulink Timing Circuit Model

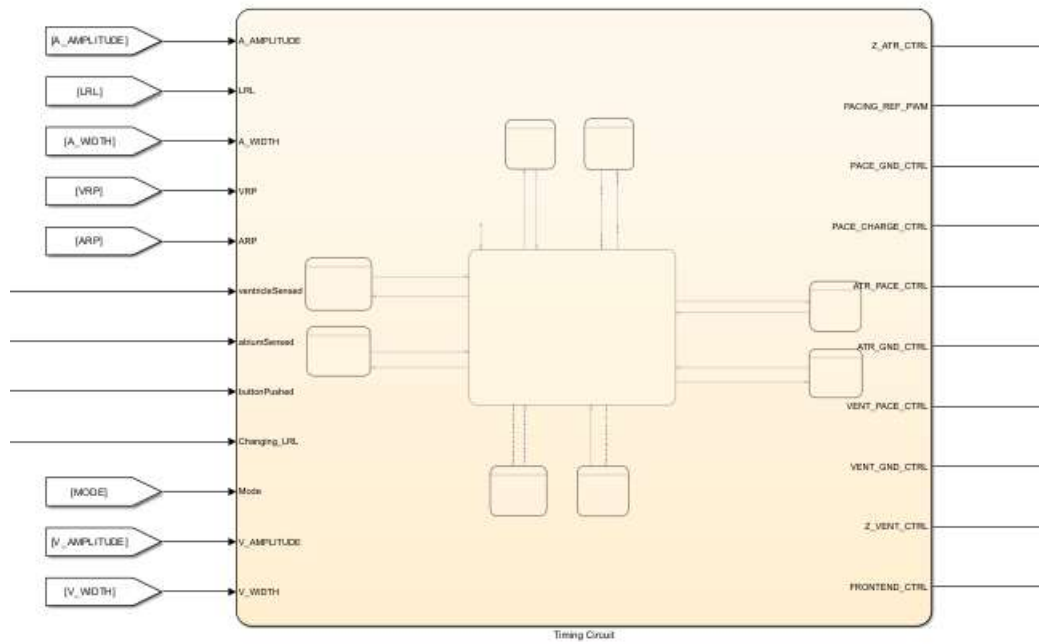


Figure 112: Timing Chart Outside

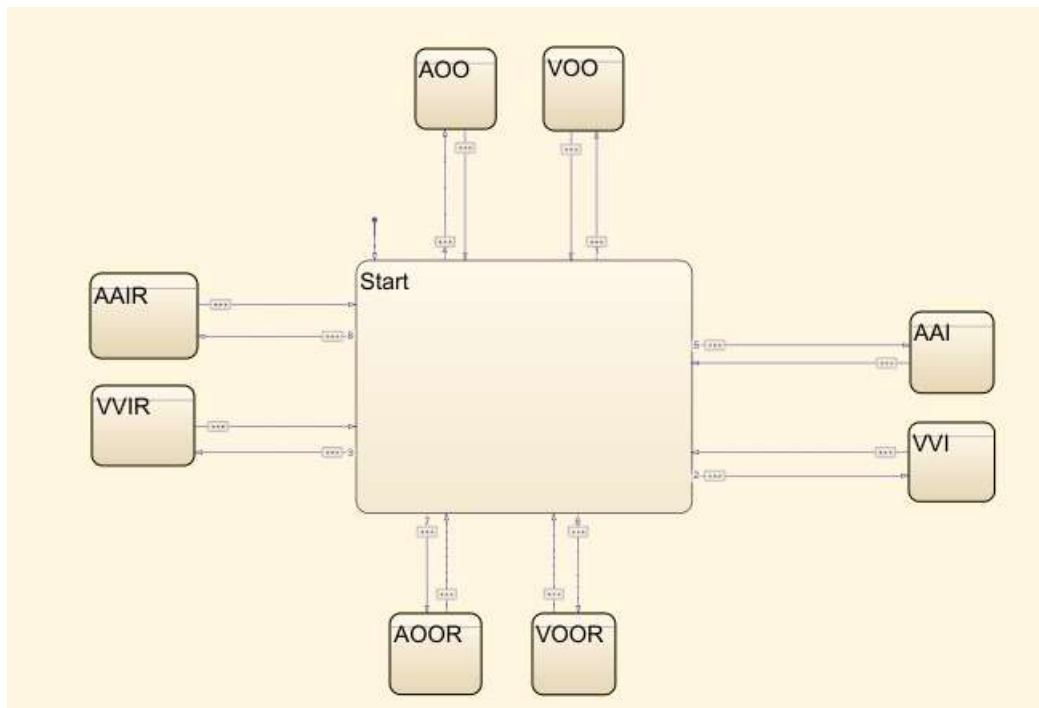


Figure 113: Timing Chart Initial State

Appendix C: Simulink Pacing Circuit Model



Figure 114: Pacing Circuit Outside

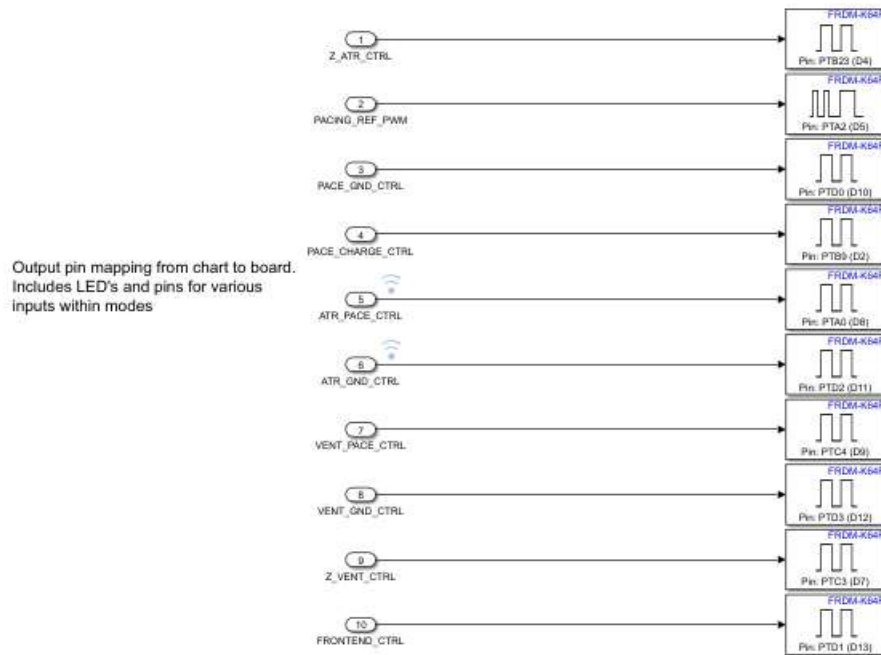


Figure 115: Pacing Circuit Inside

Appendix D: Simulink Rate Adaptive and Accelerometer Subsystem

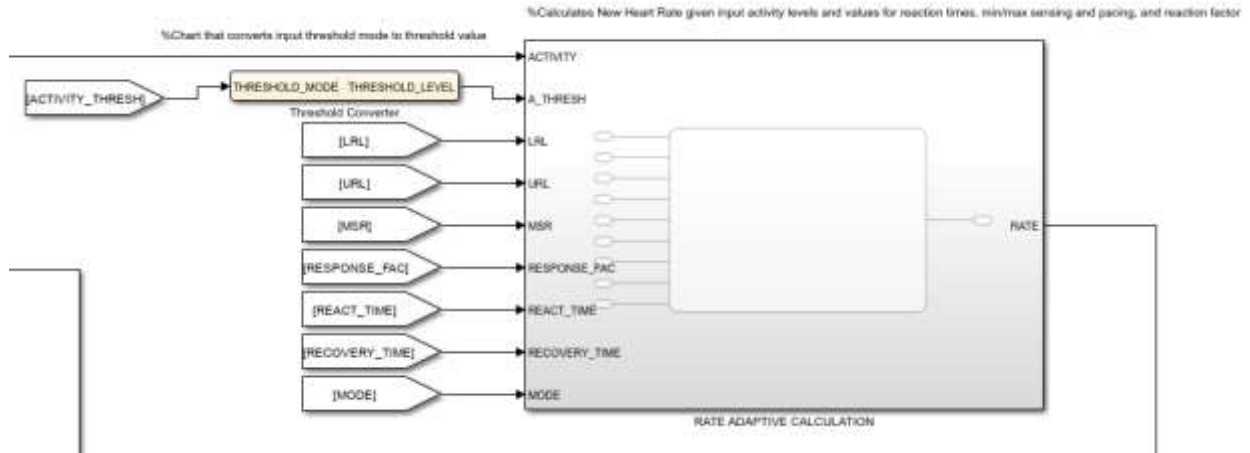


Figure 116: Rate Adaptive Exterior

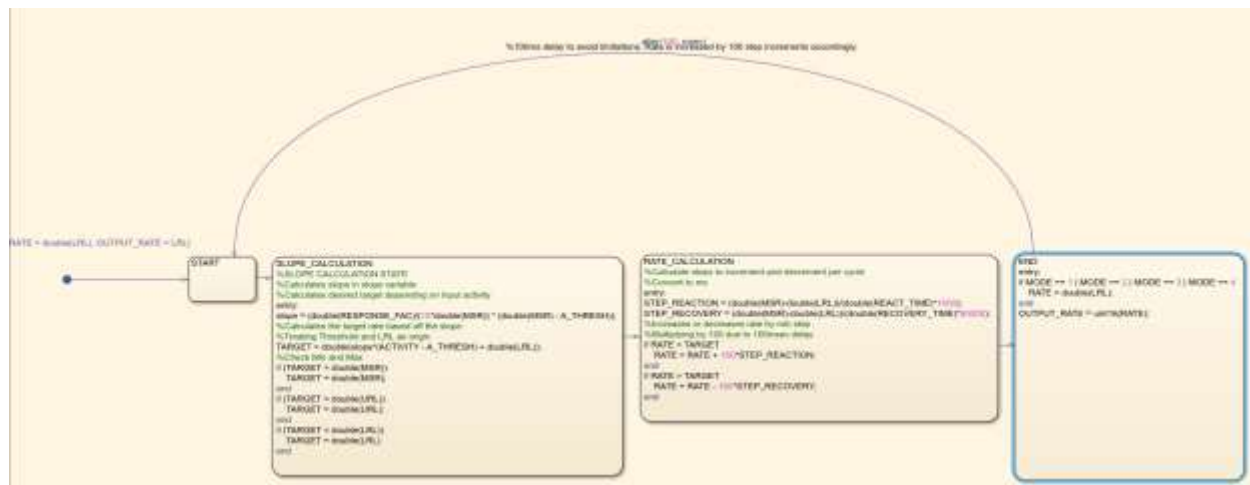


Figure 117: Rate Adaptive Calculation

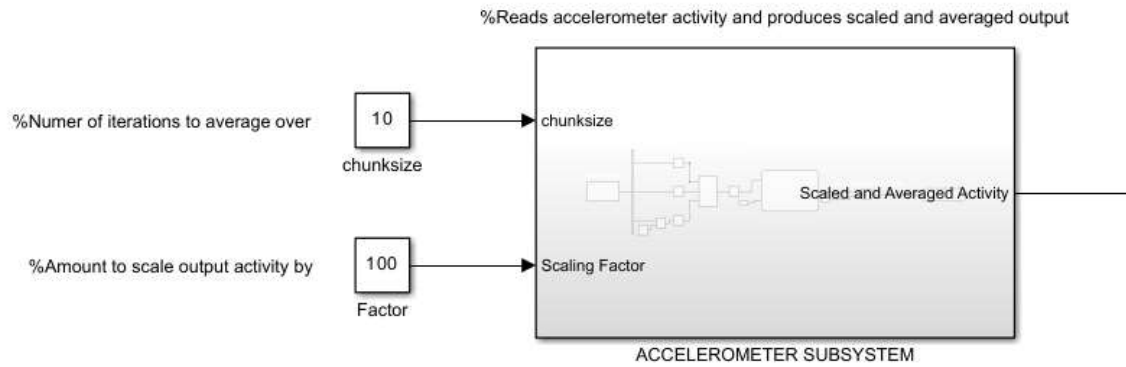


Figure 118: Accelerometer Subsystem Exterior

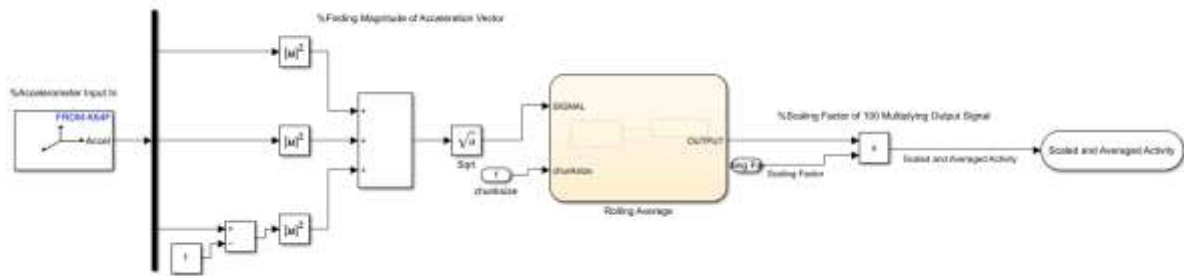


Figure 119: Accelerometer Interior

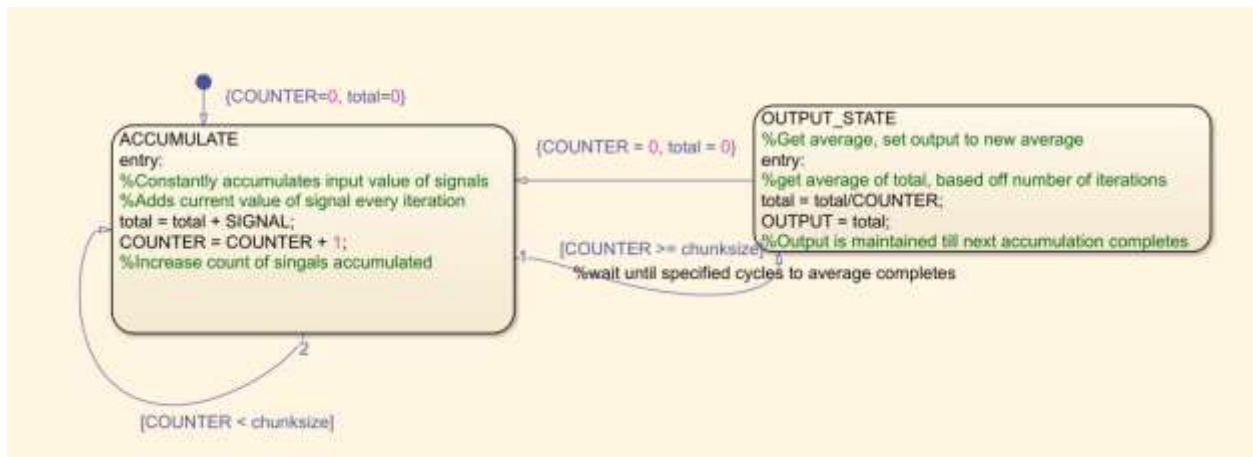


Figure 120: Scaling and Averaging Calculations

Appendix E: Serial Communication

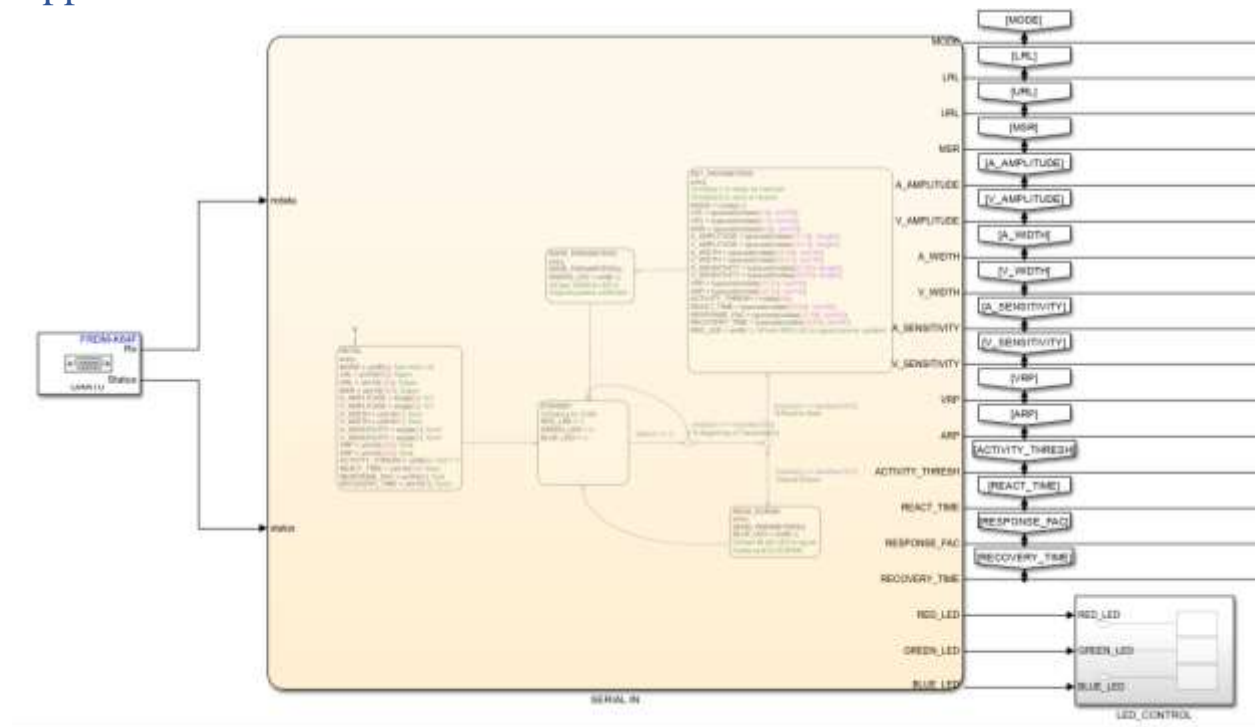


Figure 121: Serial Communication Outside

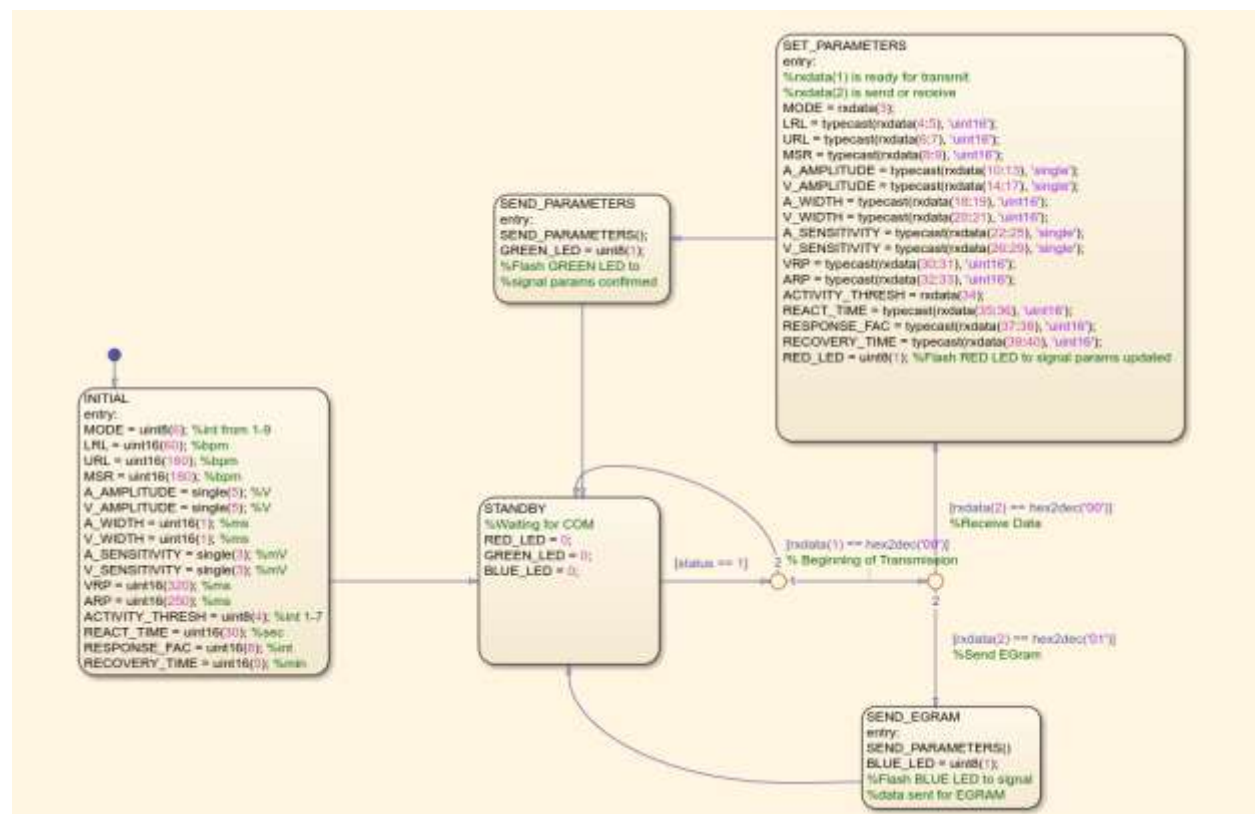


Figure 122: Serial Communication Inside

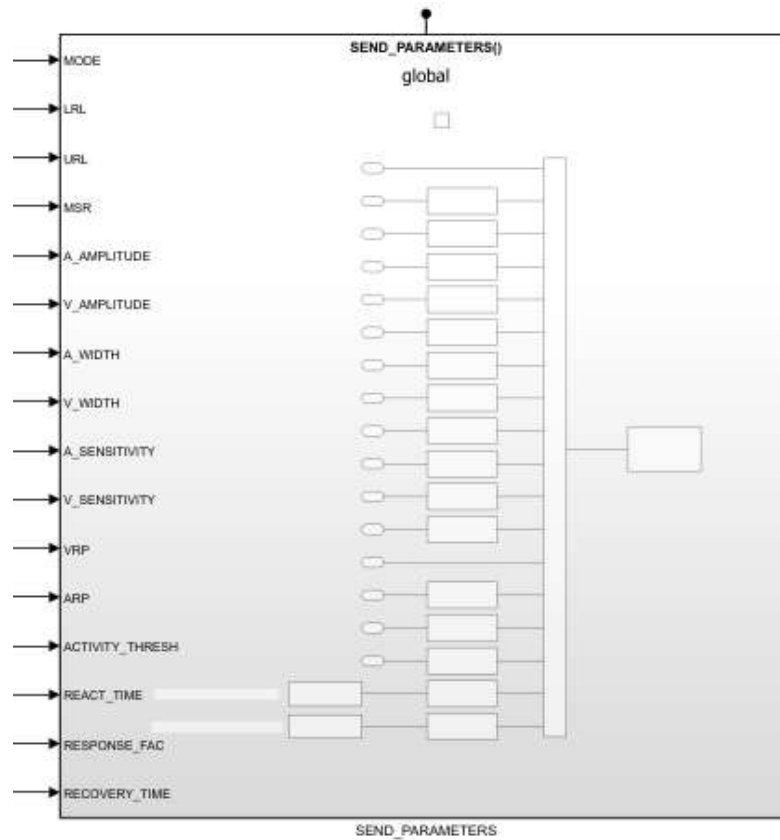


Figure 123: Serial Communication Output Outside

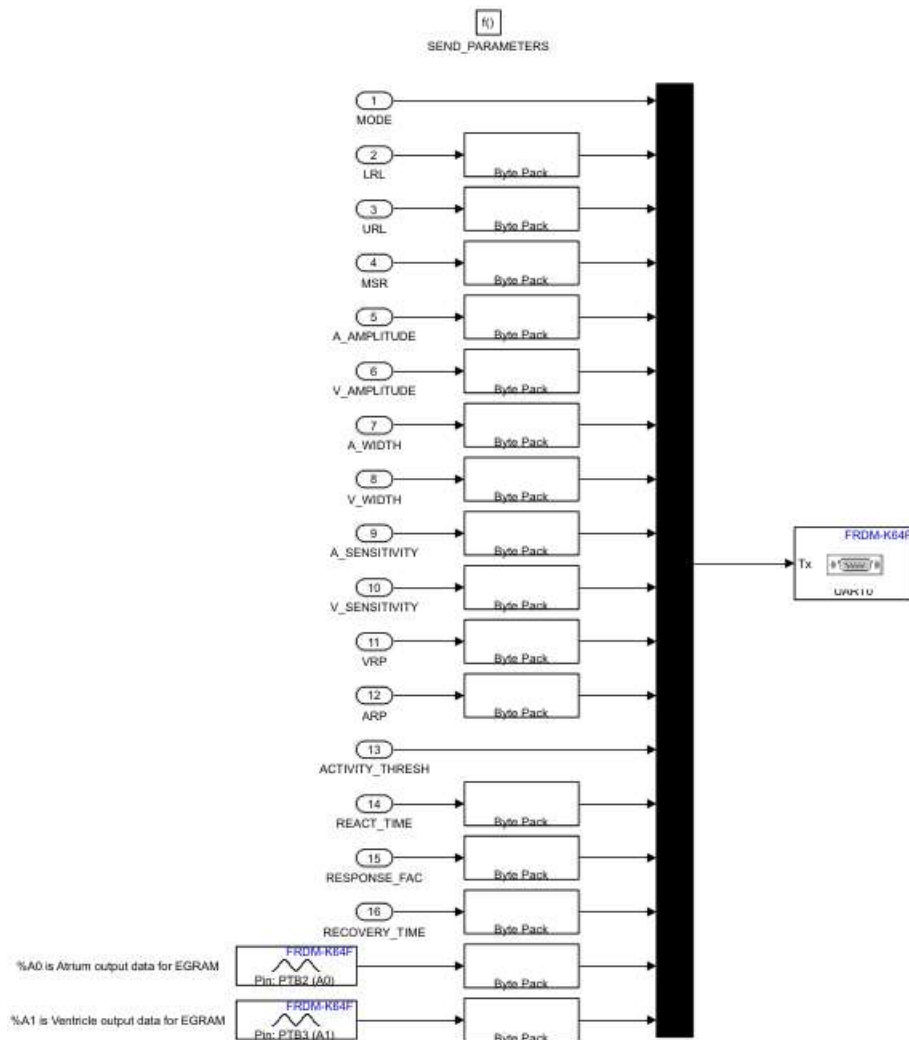


Figure 124: Serial Out and byte packaging