Assignment 2 – Documentation
MECHTRON 3K04
Fall 2023

Group 23 Members:

Zihan Wang (wangz726)

Jane D'Souza (dsouzj22)

Yuxuan Xu (xu216)

Yujia Guo (guo71)

Fan Mo (mof5)

Table of Contents

Table of Tables

Table of Figures

Part 1 – Pacemaker Design

1.1 Requirements

1.1.1 Overall Pacemaker System

The pacemaker system is composed of three primary components: the Device (pulse generator), Device Controller-Monitor (DCM), and associated Leads [1]. The pacemaker should provide dual chamber, rate adaptive bradycardia pacing support and satisfy patients' need of implantation, ambulatory, follow-up and explanation. Furthermore, the radycardia analysis capabilities should encompass pacing-related measurements like lead impedance, pacing threshold, P and R wave measurement, battery status, temporary brady pacing, motion sensor trending, and tests to be performed. The DCM serves as the primary interface for communication with the pulse generator, providing features like diagnostics and sensor history. The leads, implanted in the patient, sense cardiac electrical signal, and deliver pacing therapy. Overall, the pacemaker system is a comprehensive solution for managing cardiac conditions through non-invasive control.

1.1.2 Device Operation

The device oversees and maintains a patient's heart rate. It also identifies and administers treatment for bradycardia conditions. Furthermore, it offers customizable pacing options in both single and dual chambers, for both permanent and temporary use. When operating in adaptive rate modes, an accelerometer is utilized to gauge physical activity, thereby indicating the pacing rate for the heart. The device is programmed and communicated with through two-way telemetry from the Device Controller-Monitor (DCM), enabling physicians to make non-invasive adjustments to the device's operating mode or parameters after implantation. The device should provide history data such as output rate histograms (atrial and ventricular) and sensor output data.

1.1.3 Pulse Pacing

The device should result in pulses with programmable voltages and widths for atrial and ventricular, which provide electrical heart-pacing stimulation. Both amplitude and width for pulse pacing should be independently programmable. Bipolar electrodes and a sensing circuit are expected to facilitate rate sensing. The determination of heart rate should rely on the measured cardiac cycle data from the detected rhythm and be evaluated on a per-interval basis.

1.1.4 Operation Modes

At present stage, only 4 modes are required to be selected by the DCM system. The 4 modes are AOO, VOO, AAI and VVI. Assignment 2 is required to integrate AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR and VVIR modes all into one model.

1.1.4.1 AOO

In AOO mode the pacemaker must include a lower rate limit, an upper rate limit, atrial amplitude control, and atrial pulse width control.

1.1.4.2 VOO

In VOO mode the pacemaker must include a lower rate limit, an upper rate limit, ventricular amplitude, and ventricular pulse width.

1.1.4.3 AAI

In AAI mode the pacemaker must include a lower rate limit, an upper rate limit, atrial amplitude control, atrial pulse width control, atrial sensitivity, ARP, and PVARP.

1.1.4.4 VVI

In VVI mode the pacemaker must include a lower rate limit, an upper rate limit, ventricular amplitude, ventricular pulse width, ventricular sensitivity, and VRP.

1.1.4.5 AOOR

In AOOR mode the pacemaker must include a lower rate limit, an upper rate limit, maximum sensor rate, atrial amplitude, atrial pulse width, activity threshold, reaction time, response factor, and recovery time.

1.1.4.6 VOOR

In VOOR mode the pacemaker must include a lower rate limit, an upper rate limit, maximum sensor rate, ventricular amplitude, ventricular pulse width, activity threshold, reaction time, response factor, and recovery time.

1.1.4.7 AAIR

In AAIR mode the pacemaker must include a lower rate limit, an upper rate limit, maximum sensor rate, atrial amplitude, atrial pulse width, atrial sensitivity, ARP, PVARP, hysteresis, rate smoothing, activity threshold, reaction time, response factor, and recovery time.

1.1.4.8 VVIR

In VVIR mode the pacemaker must include a lower rate limit, an upper rate limit, maximum sensor rate, ventricular amplitude, ventricular pulse width, ventricular sensitivity, VRP, hysteresis, rate smoothing, activity threshold, reaction time, response factor, and recovery time.

1.1.4.9 DOO

In DOO mode the pacemaker must include a lower rate limit, an upper rate limit, fixed AV delay, atrial amplitude, ventricular amplitude, atrial pulse width, and ventricular pulse width.

1.1.4.8 DOOR

In DOOR mode the pacemaker must include a lower rate limit, an upper rate limit, maximum sensor rate, fixed AV delay, atrial amplitude, ventricular amplitude, atrial pulse width, ventricular pulse width, activity threshold, reaction time, response factor, and recovery time.

1.1.5 Programmable Parameters

Parameter	Programmable Values	Increment	Nominal	Tolerance
Modes	AOO	–	DDD	–
	VOO			
	AAI			
	VVI			
	AOOR			
	VOOR			
	AAIR			
	VVIR			
	DOO			
	DOOR			
Lower Rate Limit	30-50 ppm	5 ppm	60 ppm	± 8 ms
	50-90 ppm	1 ppm		
	90-175 ppm	5 ppm		
Upper Rate Limit	50-175 ppm	5 ppm	120 ppm	± 8 ms
Maximum Sensor Rate	50-175 ppm	5 ppm	120 ppm	± 4 ms
Fixed AV Delay	70-300 ms	10 ms	150 ms	± 8 ms
A or V Pulse Amplitude Regulated	Off, 0.5 – 3.2 V	0.1 V	3.5V	12 %
	3.5 – 7.0 V	0.5 V		
A or V pulse Width	0.05 ms	–	0.4 ms	0.2 ms
	0.1 – 1.9 ms	0.1 ms		
A or V Sensitivity	0.25, 0.5, 0.75	–	A-0.75 mV	± 20%
	1.0-10 mV	0.5 mV	V-2.5 mV	
Ventricular Refractory Period	150 – 500 ms	10 ms	320 ms	± 8 ms
Atrial Refractory Period	150 – 500 ms	10 ms	250 ms	± 8 ms
PVARP	150-500 ms	10 ms	250 ms	± 8 ms
Hysteresis Rate Limit	Off or same choices as LRL	–	Off	± 8 ms
Rate Smoothing	Off, 3, 6, 9, 12, 15, 18, 21, 25%	–	Off	± 1%
Activity Threshold	V-Low, Low, Med-Low, Med, Med-High, High, V-High	–	Med	cc
Reaction Time	10-50 sec	10 sec	30 sec	± 3 sec
Response Factor	1-16	1	8	–

Recovery Time	2-16 min	1 min	5 min	± 30 sec

Table 1: Simulink Programable Parameters

1.2 Design Decisions

A subsystem block is used to map all the hardware output pins for use in the program to their names as defined in Table 1 of Pacemaker Shield Explained. Another subsystem is used to map the hardware input pins to their names in the same table. We also created a subsystem block which contains all the Serial Input ports for future communication with the DCM system, although this subsystem block is not used for assignment 1. This allows the variables we used within the program to be more readable. 4 state charts were used for the 4 operation modes that are required and a multiport switch is used for mode selection. Gains of 20 were used to convert the input A or V amplitude from the voltage number between 0 to 5 volts to a PWM duty cycle number between 0 to 100. Annotation is added to all the blocks using the bolded red font to make the whole Simulink model more readable.

1.3 Simulink Diagram

The Simulink diagram must include necessary annotation to understand the model. There should be screenshots of the Simulink model in the documentation.

AOO mode:

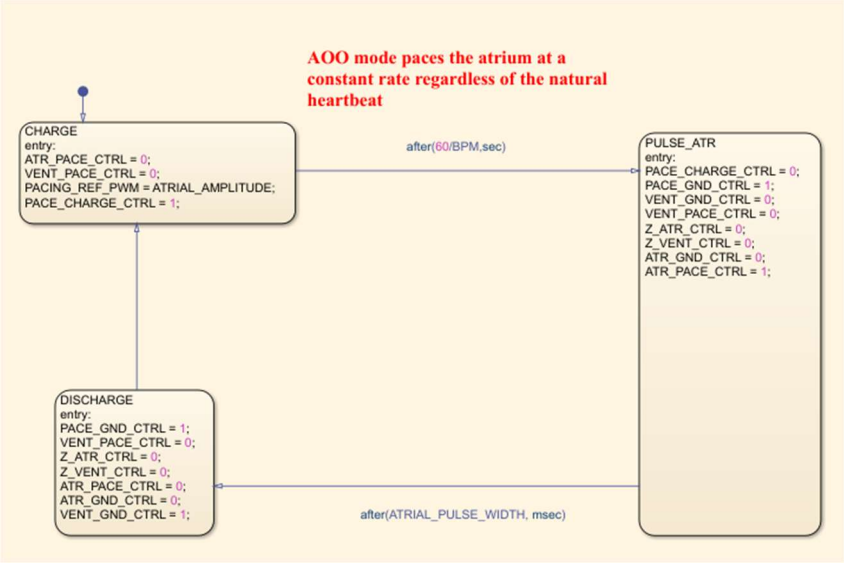


Figure 1: AOO mode Stateflow

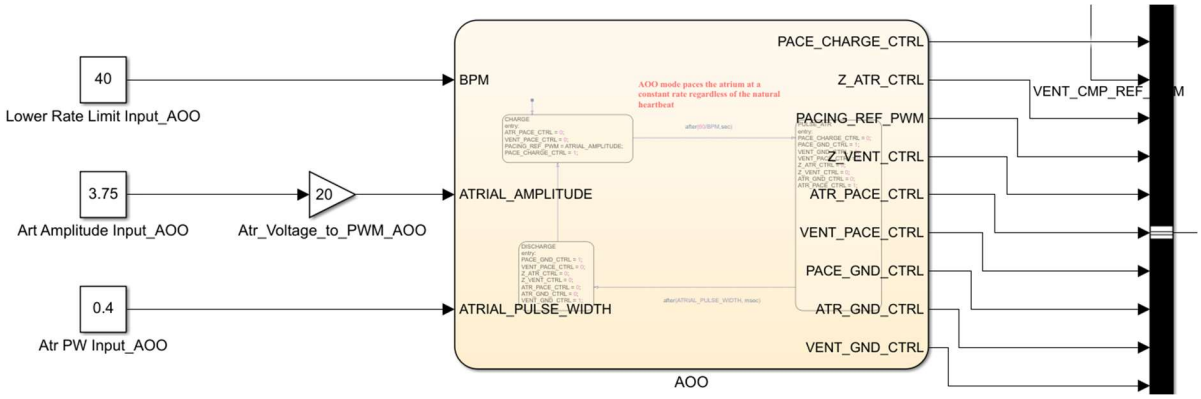


Figure 2: Inputs of AOO mode

These parameters (BPM, ATRIAL_AMPLITUDE, and ATRIAL_PULSE_WIDTH) are programmed with specific numerical values and are used in the charging and discharging processes of the pacemaker, which delivers the appropriate electrical signals to the hardware outputs. To charge C22, PACE_CHARGE_CTRL is set to 1 to prepare the pacing operation. Once the pacing capacitor is charged (60/BPM, sec), the system requires pacing in the atrium by setting PACE_GND_CTRL and ATR_PACE_CTRL to 1. The capacitor C21 is discharged by grounding the paced chamber to ensure a net zero current and protect the patient after pacing occurs.

VOO mode:

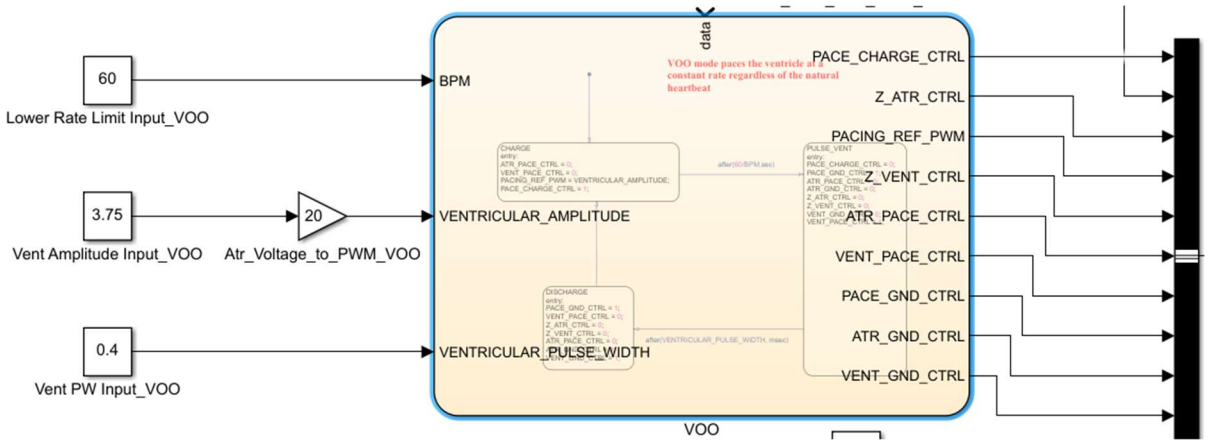


Figure 3: Inputs of VOO mode

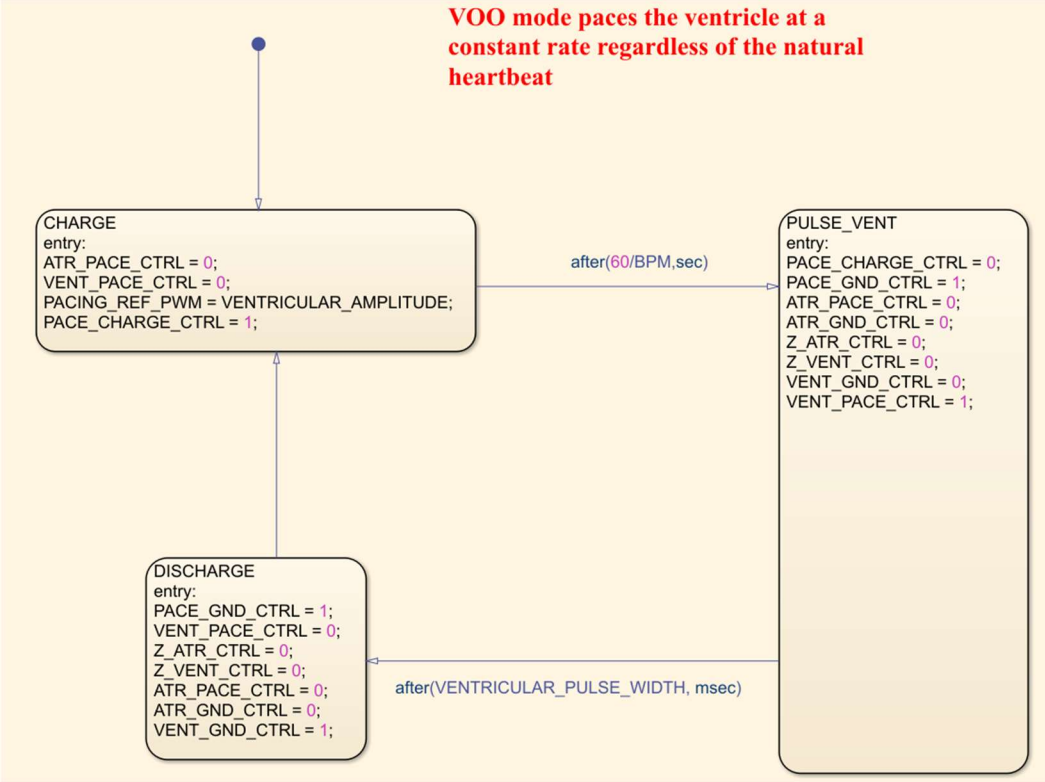


Figure 4: VOO mode Stateflow

The process of VOO mode is similar to the process of AOO mode except for locations of chamber. These parameters and inputs of VOO mode are the same as the AOO modes. In VOO mode, both the pacing signal and the sensing signal occur in the heart's ventricles, which emits a consistent signal at fixed intervals to stimulate the ventricles and maintain rhythm. Additionally, if the ventricles can generate their own heartbeat signals, the pacemaker stops from delivering pacing pulses.

AAI mode:

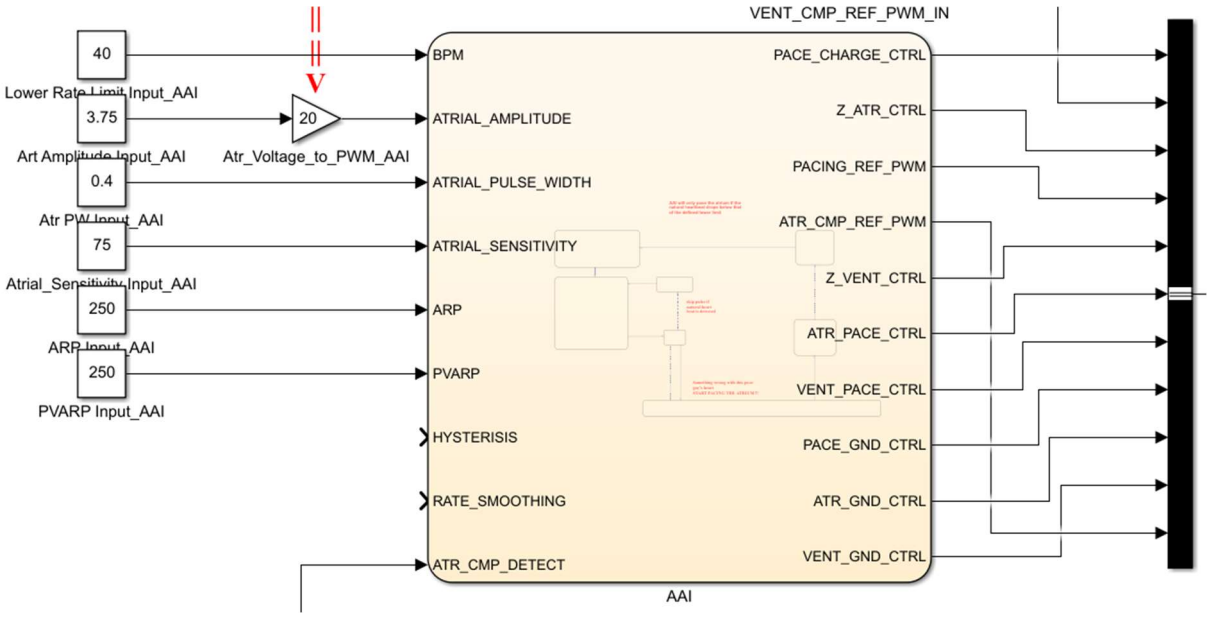


Figure 5: Inputs of AAI mode

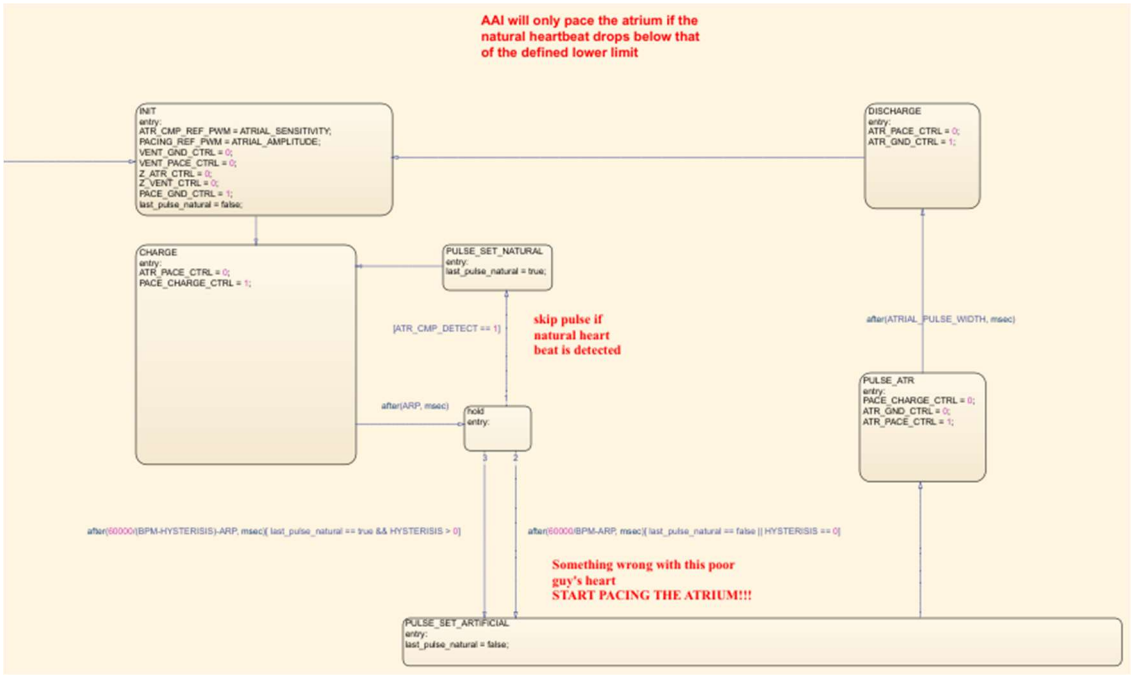


Figure 6: AAI mode Stateflow

At the initial state, the system set parameters of ATRIAL_SENSITIVITY and ATRIAL_AMPITUDE and preset the heartbeat is not produced naturally by the atrium (last_pulse_natural = false). The pacemaker system starts charging process (PACE_CHARGE_CTRL =1) to prepare for the next atrial pacing. After we determine the last heartbeat that was generated by the pacemaker or patients' atrium, the system initiates atrial pacing if last_pulse_natural = false and discharges to release capacitive charge (ATR_GND_CTRL =1).

VVI mode:

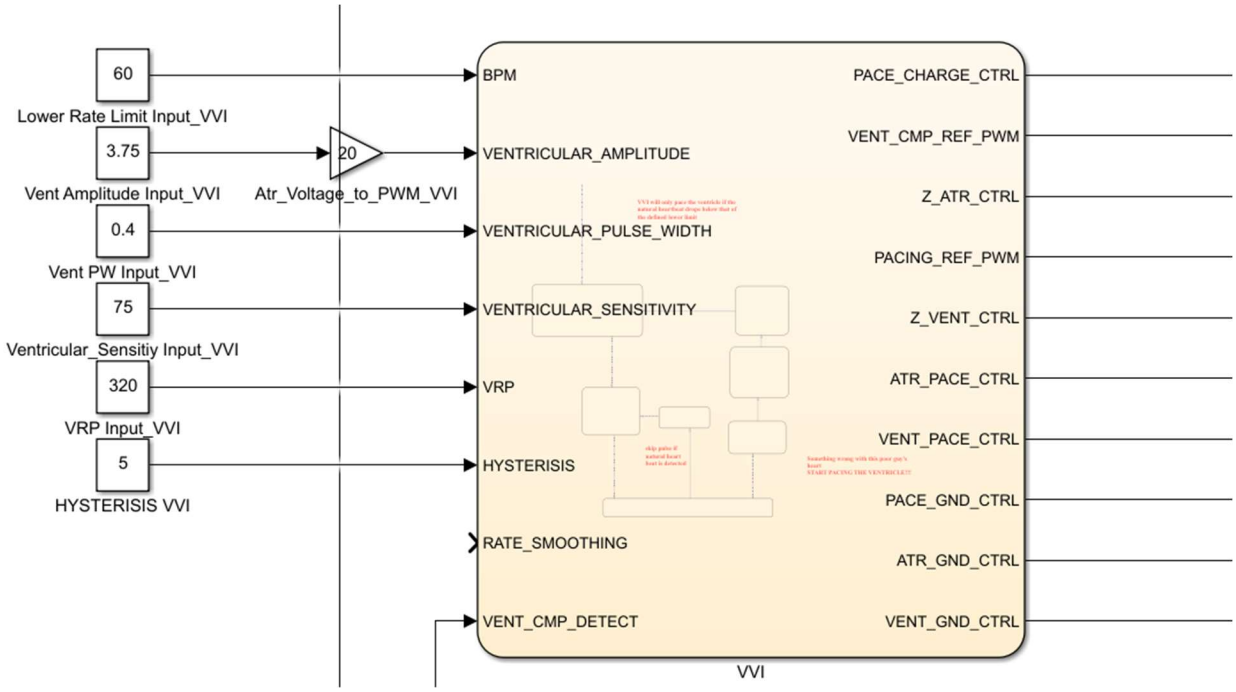


Figure 7: Inputs of VVI mode

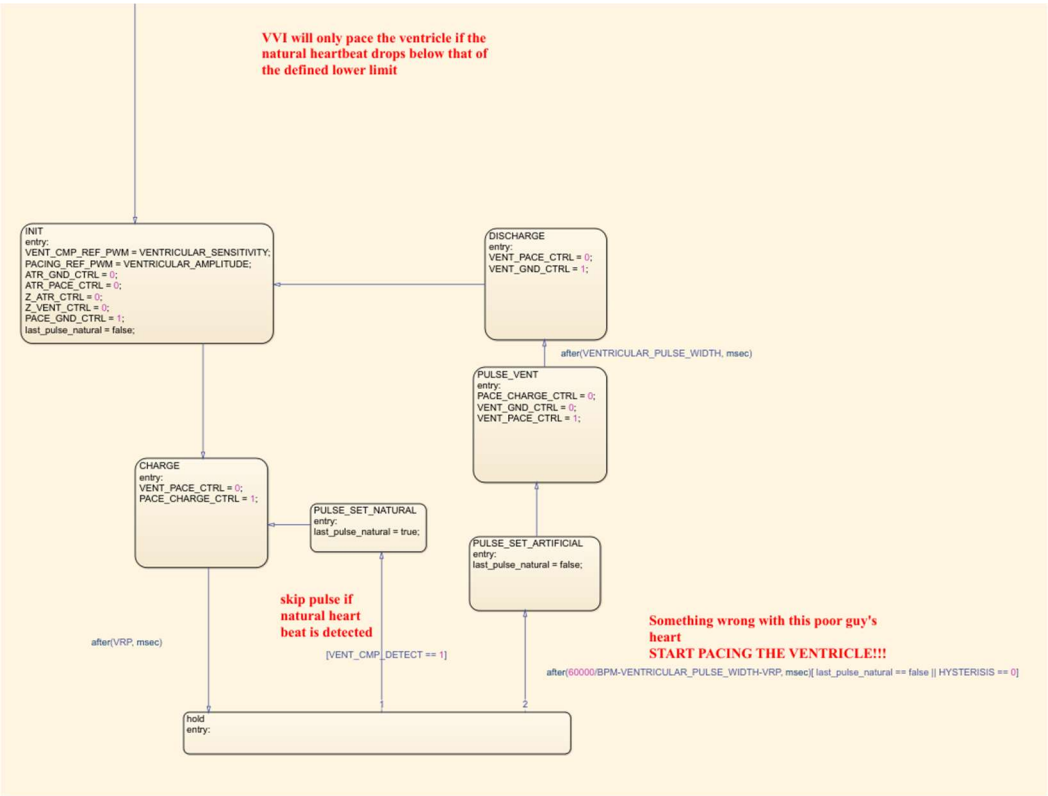


Figure 8: VVI mode Stateflow

The process of VVI mode is similar to the process of AAI mode. The main difference between these two modes is that the pacemaker primarily paces the ventricles to maintain the rhythm in VVI mode.

Change the mode:

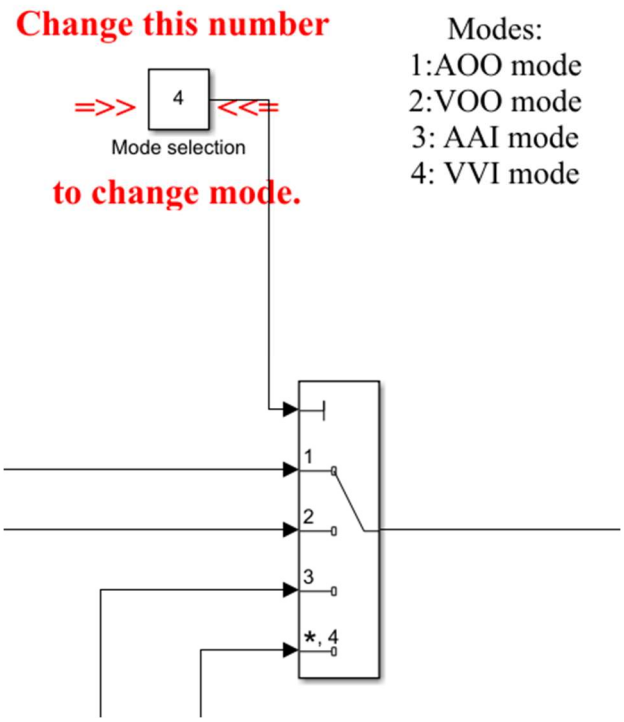


Figure 9: Mode Selection Button

The mode selection button is set to switch between pacemaker modes (AOO, VOO, AAI and VVI modes) ranging from 1 to 4.

Adaptive Rate modify:

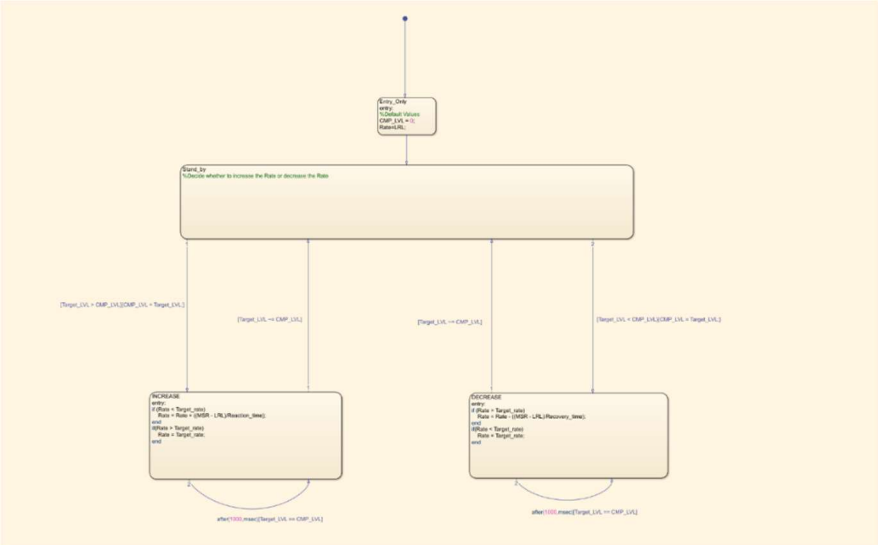


Figure 10: Adaptive Rate Modify

The additional 4 modes in Assignment 2, AOOR, VOOR, AAIR and VVIR, requires real pacing rate calculations since the pacing period in these modes need to be controlled by an adaptive rate instead of lower rate limit.

AOOR mode:

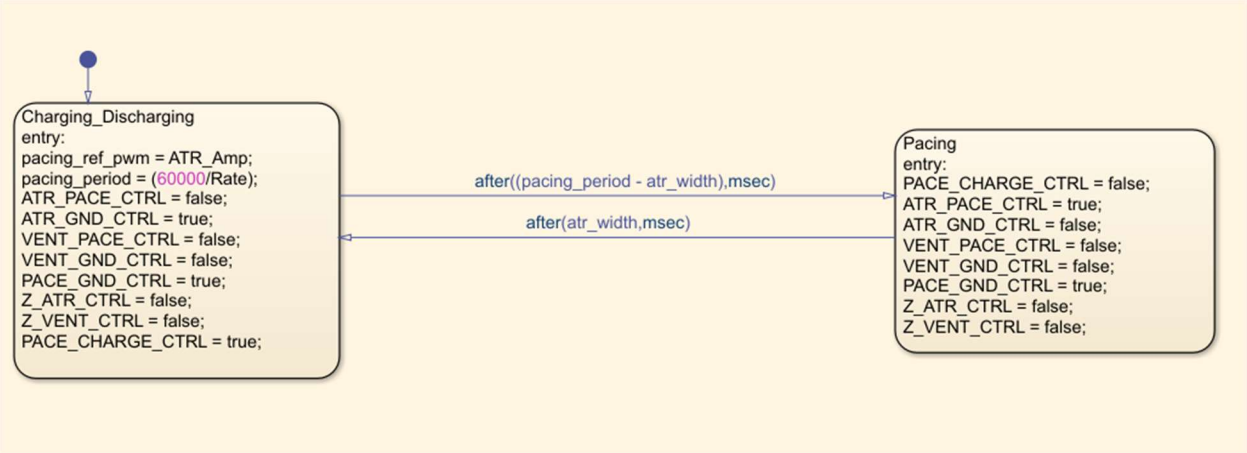


Figure 11: AOOR mode Stateflow

The AOOR mode is composed of charging/discharging state and pacing state. The duration of the charging state is determined by the 'atr_width' variable and is subsequently transferred to the pacing state. Following capacitor charging, the pacing state involves cardiac pacing. The main difference between AOO and AOOR mode is the pacing period, with AOOR being controlled by an adaptive rate instead of lower rate limit.

VOOR mode:

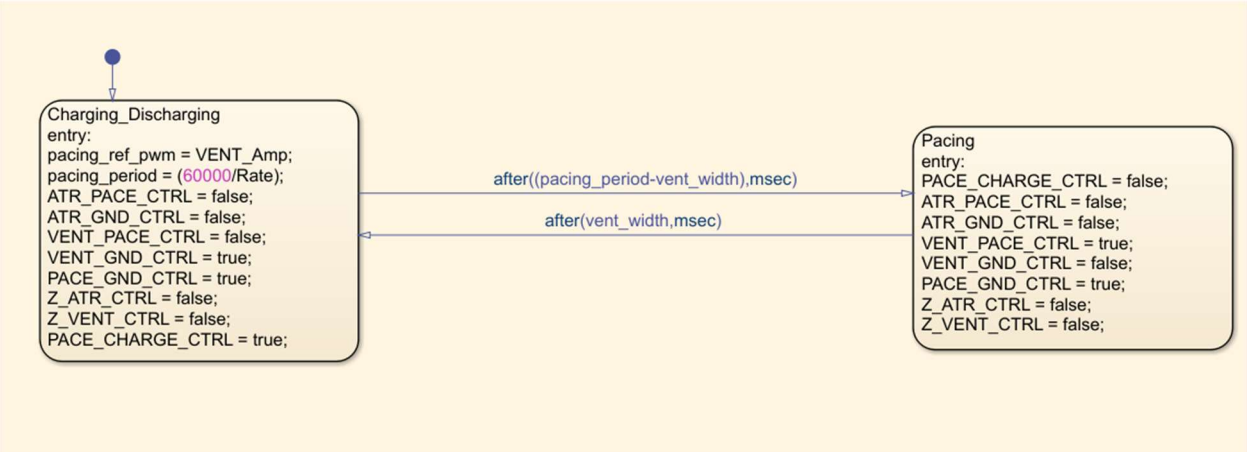


Figure 12: VOOR mode Stateflow

The process of the VOOR mode is similar to the AOOR mode. The primary distinction between these two modes lies in the determination of the charging state, where the duration is governed by the 'vent_width' variable. Furthermore, VOOR mode is associated with the ventricle (VENT_GND_CTRL), while AOOR mode is linked with the atrium (ATR_GND_CTRL).

AAIR mode:

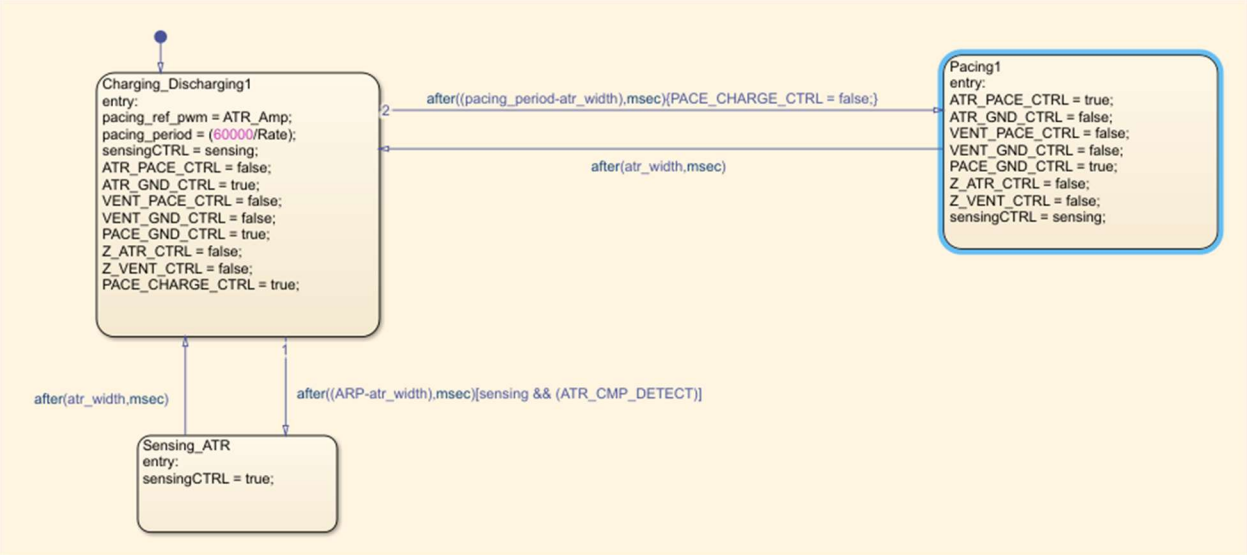


Figure 13: AAIR mode Stateflow

The AAIR mode is composed of Charging_Discharging state, sensing_ATR state and Pacing state. Charging_Discharging state supports the charging capacitor to the pacing state and discharge after pacing state. If sensing natural heartbeat (ATR_CMP_DETECT), pacing is followed by natural hearbeats. After charging capacitor and time period for natural hearbeat, the pacemaker executes the pacing state. The logic of AAI mode is similar to the AAIR mode, but the pacing period in AAIR is determined by the adaptive rate rather than lower rate limit.

VVIR mode:

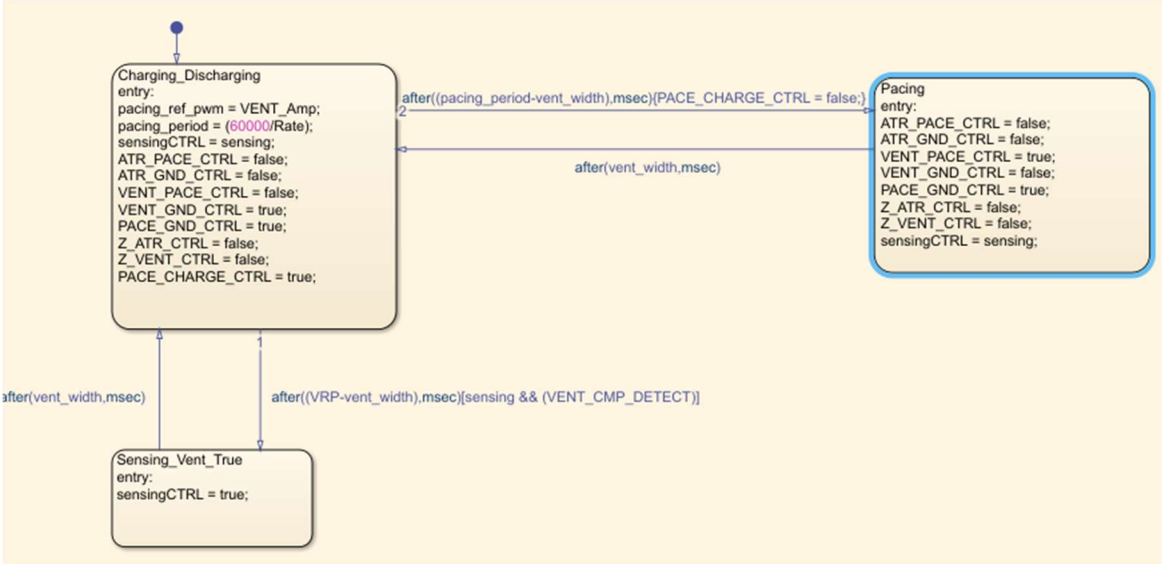


Figure 14: VVIR mode Stateflow

Both VVIR and AAIR modes follow a similar process, which includes all three states. In VVIR mode, sensing involves monitoring the amplitude of natural heartbeats in the ventricle instead of atrial amplitude in AAIR mode. Additionally, VVIR mode executes pacing in the ventricle, delivering pacing pulses to the ventricle.

DOO mode:

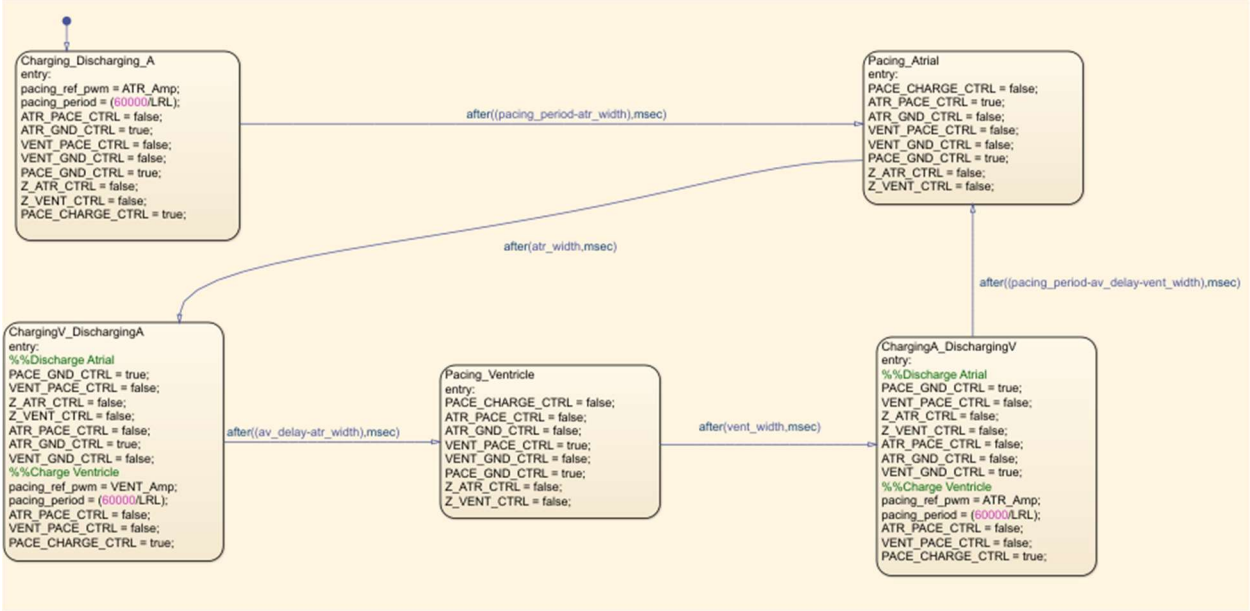


Figure 15: DOO mode Stateflow

The DOO mode comprises five states. The Charging_Discharging states are responsible for preparing the capacitor for atrial pacing. After a specified time period, the pacemaker executes atrial pacing in the Pacing_Atrial state. The ChargingV_DischargingA state discharges the atrium and prepares for ventricular charging. After ventricular pacing, the pacemaker discharges the ventricle and charges the atrium. Thus, the pacemaker cycles through these four states in a continuous loop.

DOOR mode:

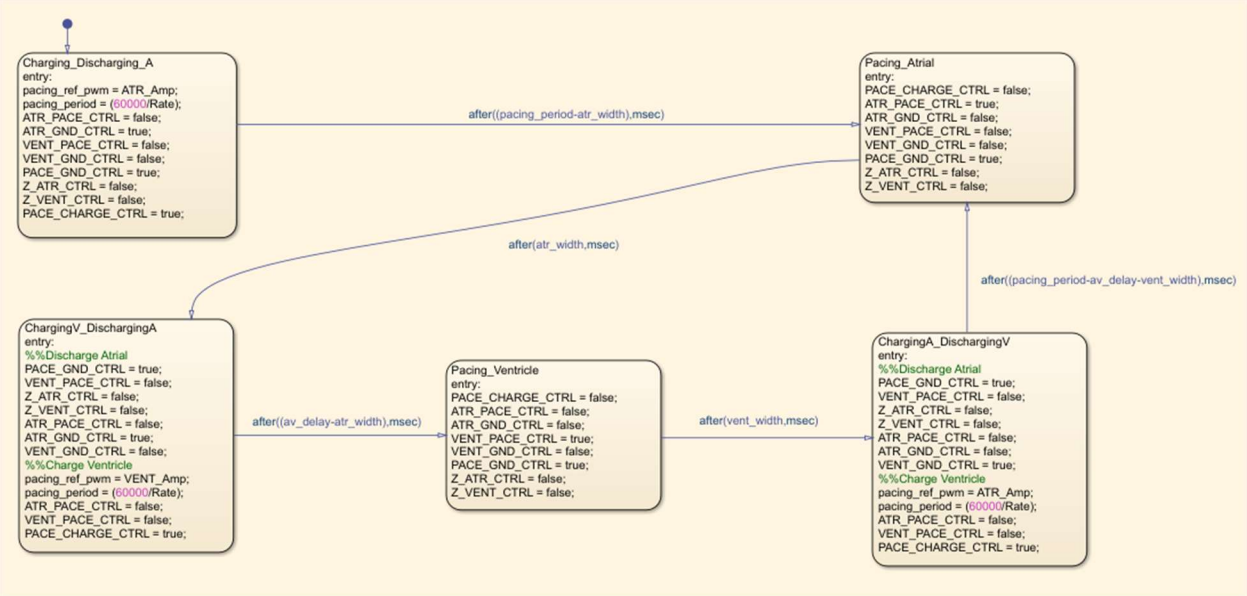
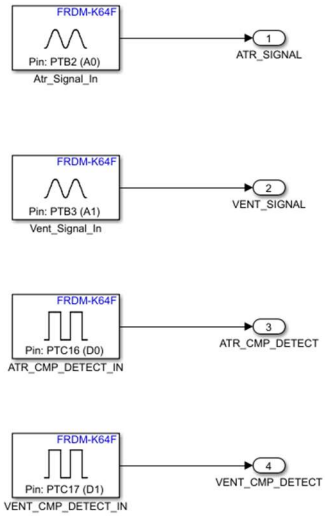


Figure 16: DOOR mode Stateflow

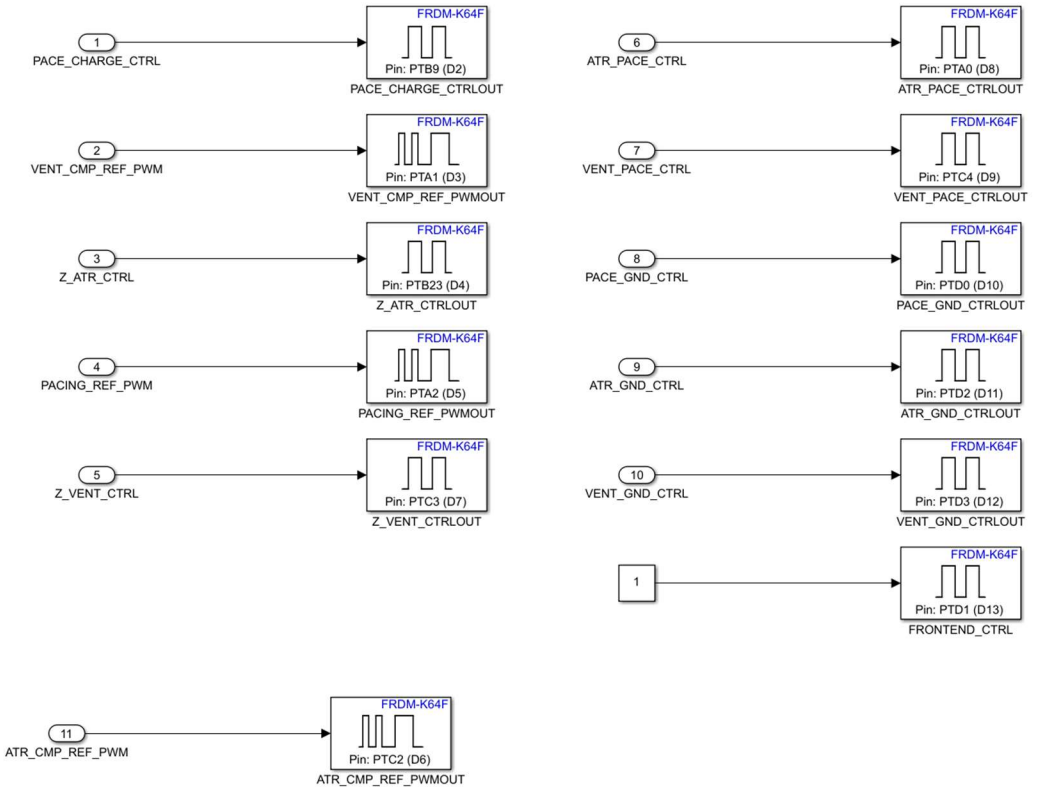
The DOOR mode follows a logic similar to the DOO mode but is determined by an adaptive pacing rate rather than LRL.

Hardware inputs:



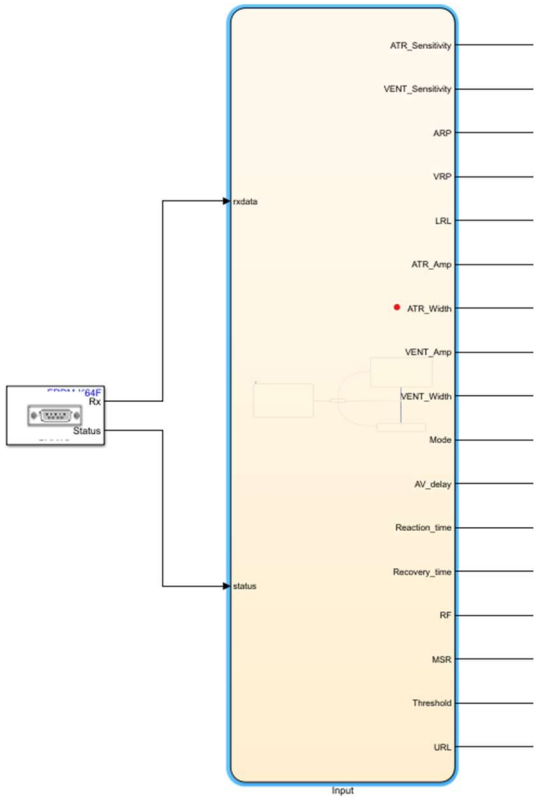
These parameters read digital signals from the hardware and deliver them to the different modes as inputs. Each parameter is assigned pin based on the pin map of the K64F microcontroller.

Hardware outputs:

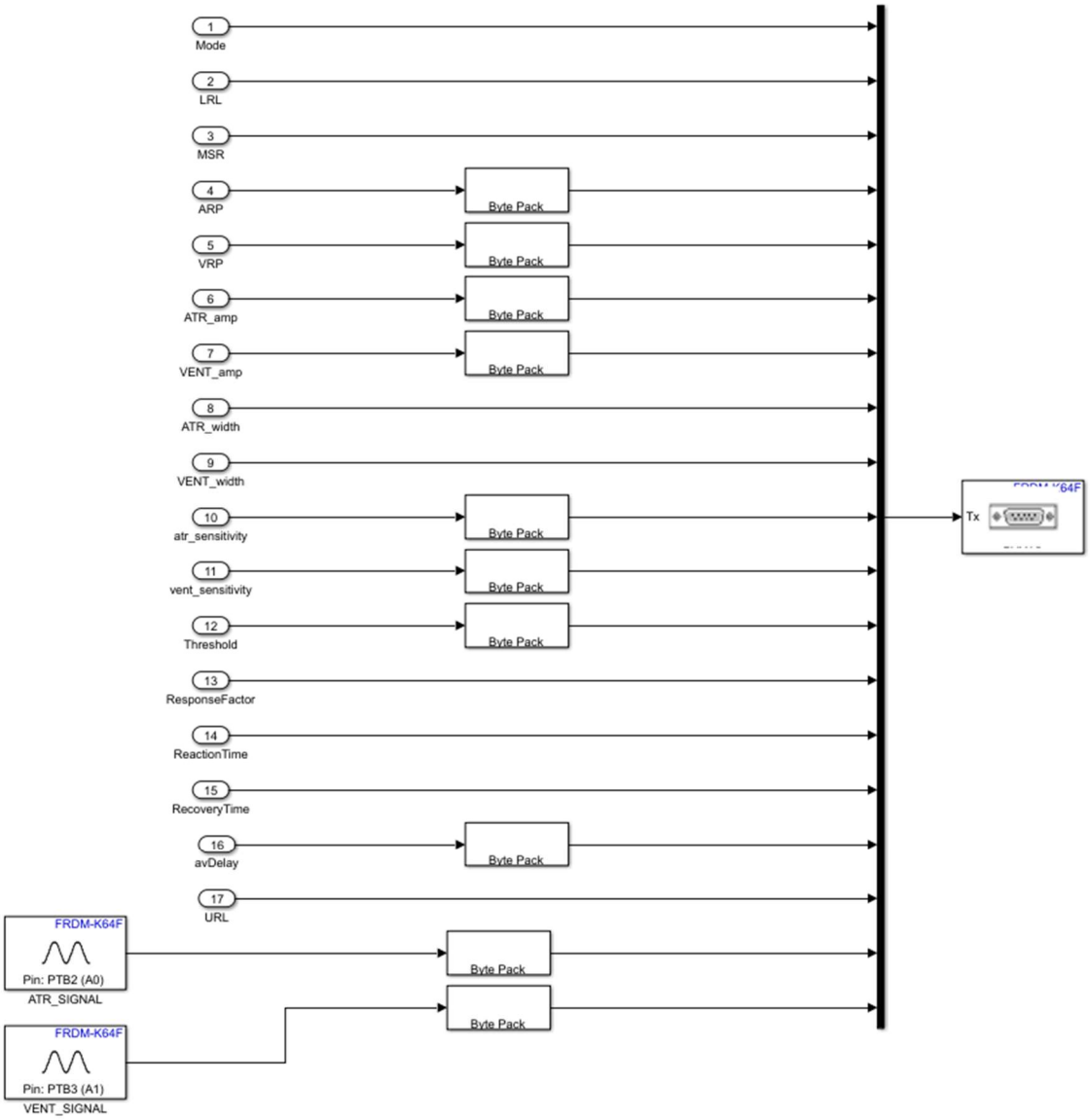


The output data corresponds to the pins on the hardware, similar to the handling of input data. This separation allows for a clear situation between signal processing in the modes and the hardware.

Serial Communications:



The inputs are assigned via serial communications.



The block is able to send the value back to the DCM. The ATR_SIGNAL and the VENT_SIGNAL is required to plot the egram while the rest data were sent to make sure the data is correctly sent into the pacemaker by DCM correctly through UART transmission.

1.4 Testing and Results

AOO Test Cases:

Report ID: AOO1

Active Test Routine

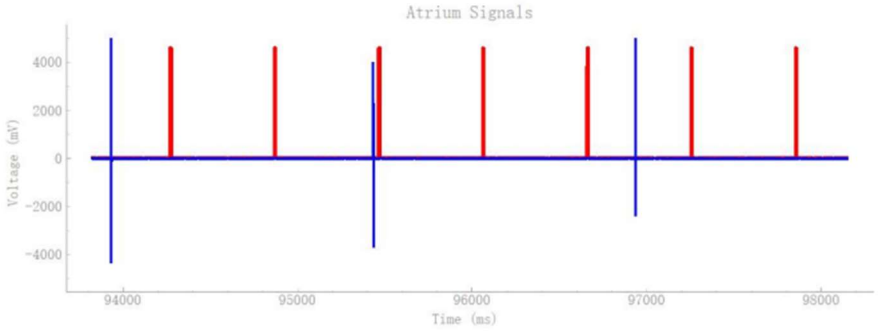
Atrium PW: 11.0 ms

Ventricular PW: 0.0 ms

Heart Rate: 100 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

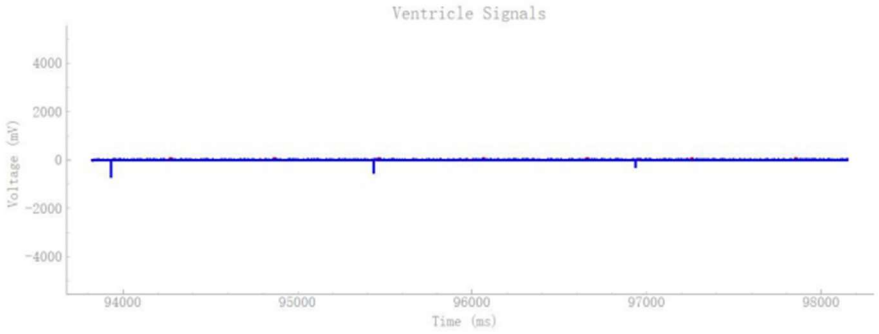


Figure 17: AOO Test Case 1

Condition: Patient with natural heartbeat

Expected Output: Pacemaker paces with atrium

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

Report ID: AOO2

Active Test Routine

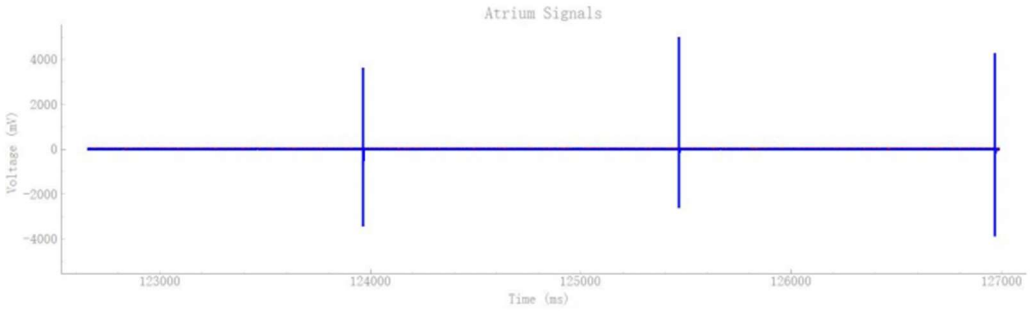
Atrium PW: 0.0 ms

Ventricular PW: 0.0 ms

Heart Rate: 30 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

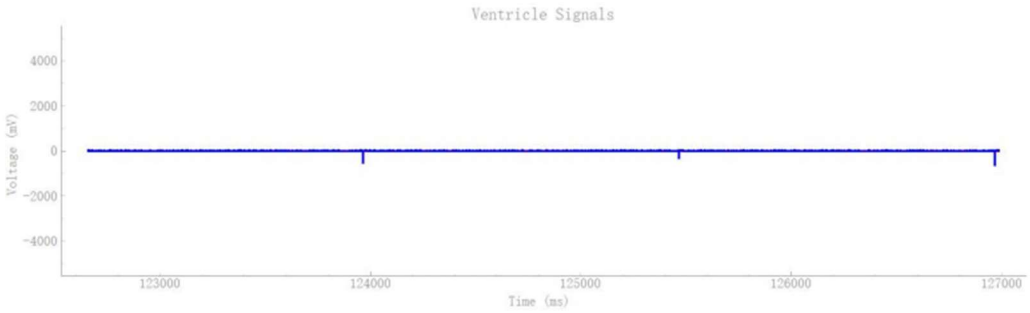


Figure 18: AOO Test Case 2

Condition: Patient without natural heartbeat

Expected Output: Pacemaker paces with atrium

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

VOO Test Cases:

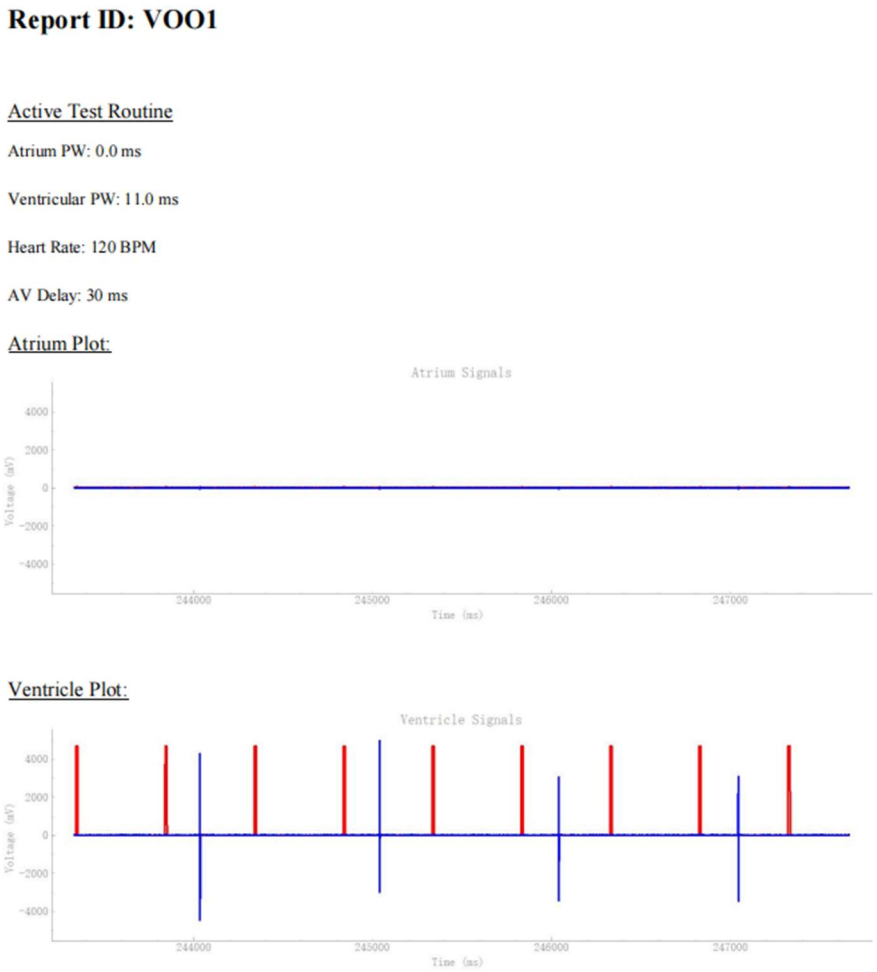


Figure 19: VOO Test Case 1

Condition: Patient with natural heartbeat

Expected Output: Pacemaker paces with ventricle

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

Report ID: VOO2

Active Test Routine

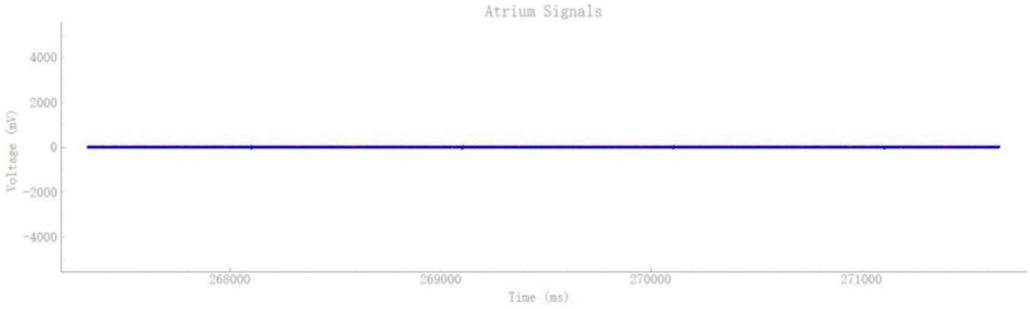
Atrium PW: 0.0 ms

Ventricular PW: 0.0 ms

Heart Rate: 120 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

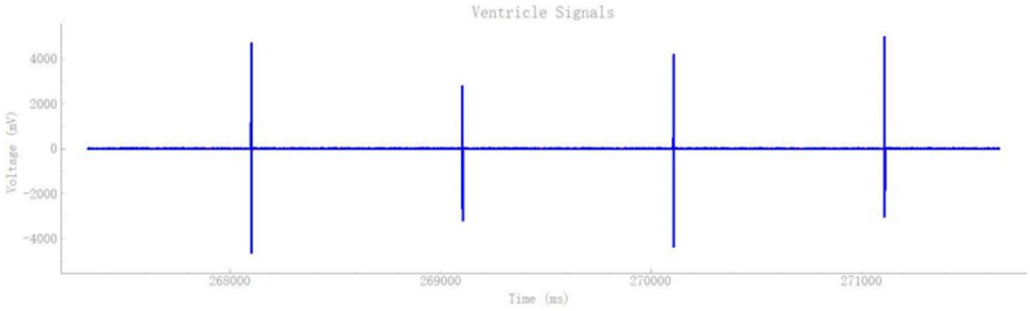


Figure 20: VOO Test Case 2

Condition: Patient without natural heartbeat

Expected Output: Pacemaker paces with ventricle

Actual Output: Pacemaker paces as expected

Pass/Fail: Pass

AAI Test Cases:

Report ID: AAI1

Active Test Routine

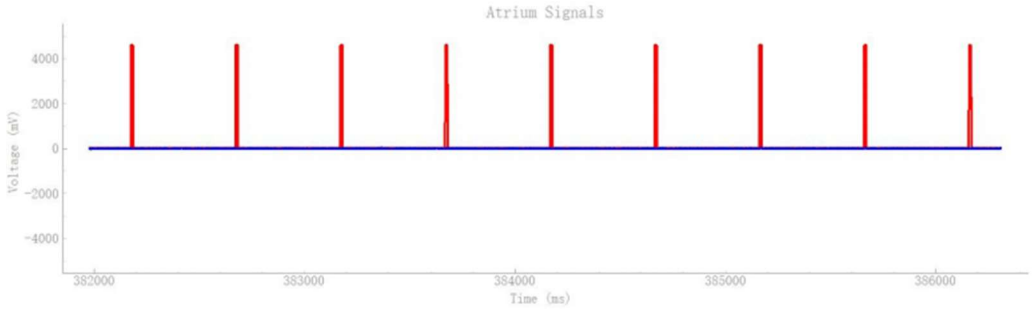
Atrium PW: 11.0 ms

Ventricular PW: 0.0 ms

Heart Rate: 120 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

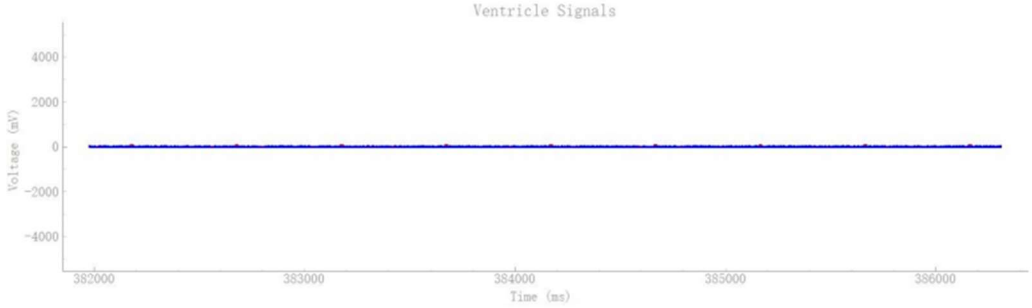


Figure 21: AAI Test Case 1

Condition: Patient with natural heartbeat

Expected Output: Pacemaker do not pace with atrium

Actual Output: Pacemaker do not pace with atrium

Pass/Fail: Pass

Report ID: AAI2

Active Test Routine

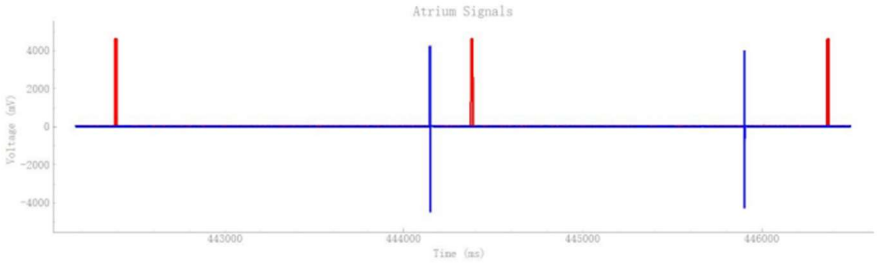
Atrium PW: 11.0 ms

Ventricular PW: 0.0 ms

Heart Rate: 30 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

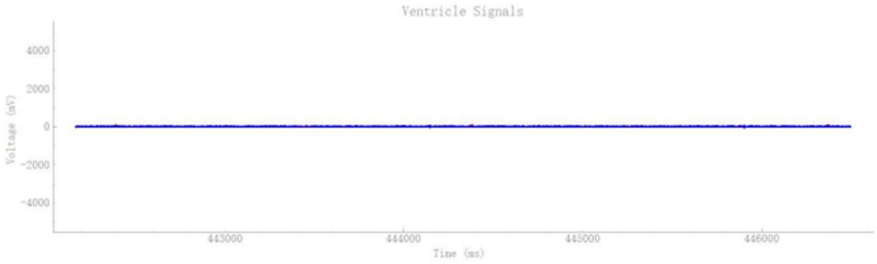


Figure 22: AAI Test Case 2

Condition: Patient with low frequency heartbeat

Expected Output: Pacemaker paces with atrium

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

Report ID: AAI3

Active Test Routine

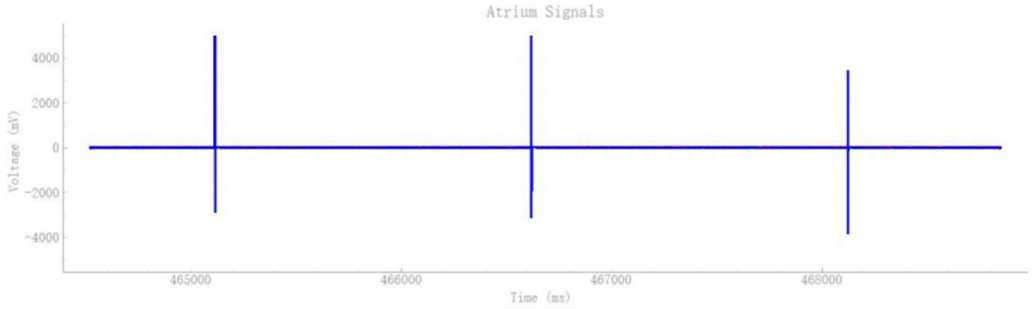
Atrium PW: 0.0 ms

Ventricular PW: 0.0 ms

Heart Rate: 30 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

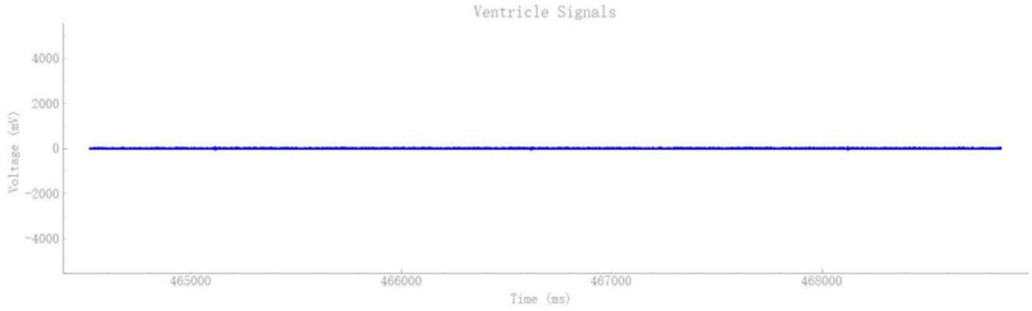


Figure 23: AAI Test Case 3

Condition: Patient without natural heartbeat

Expected Output: Pacemaker paces with atrium

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

VVI Test Cases:

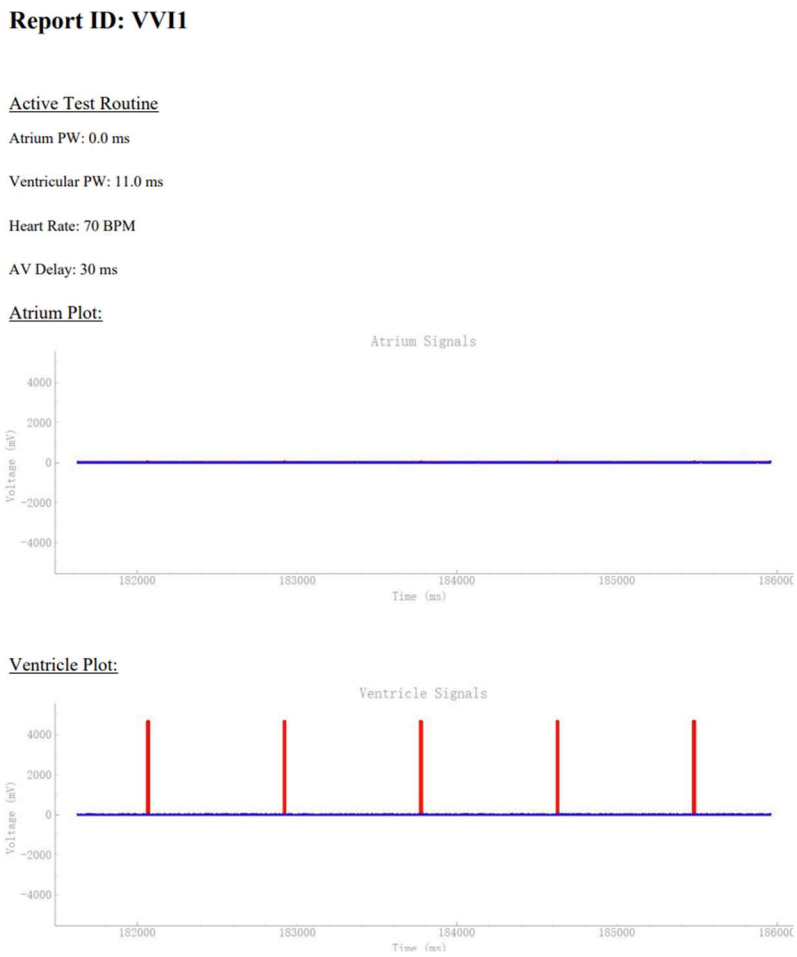


Figure 24: VVI Test Case 1

Condition: Patient with natural heartbeat

Expected Output: Pacemaker do not pace with ventricle.

Actual Output: Pacemaker do not pace with ventricle

Pass/Fail: Pass

Report ID: VVI2

Active Test Routine

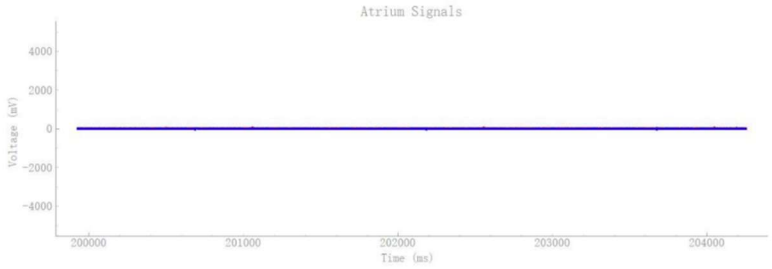
Atrium PW: 0.0 ms

Ventricular PW: 11.0 ms

Heart Rate: 40 BPM

AV Delay: 30 ms

Atrium Plot:



Ventricle Plot:

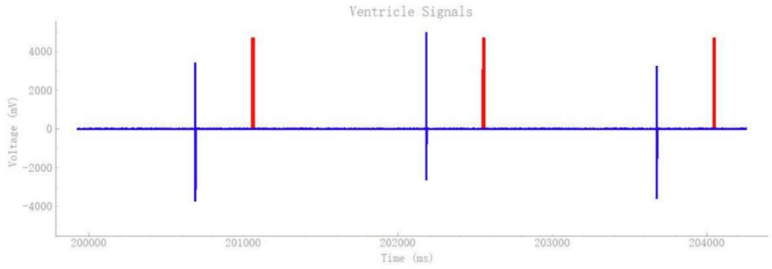


Figure 25: VVI Test Case 2

Condition: Patient with low frequency heartbeat

Expected Output: Pacemaker paces with ventricle.

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

Figure 26: VVI Test Case 3

Condition: Patient without natural heartbeat

Expected Output: Pacemaker paces with ventricle.

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

AAIR Test Cases:

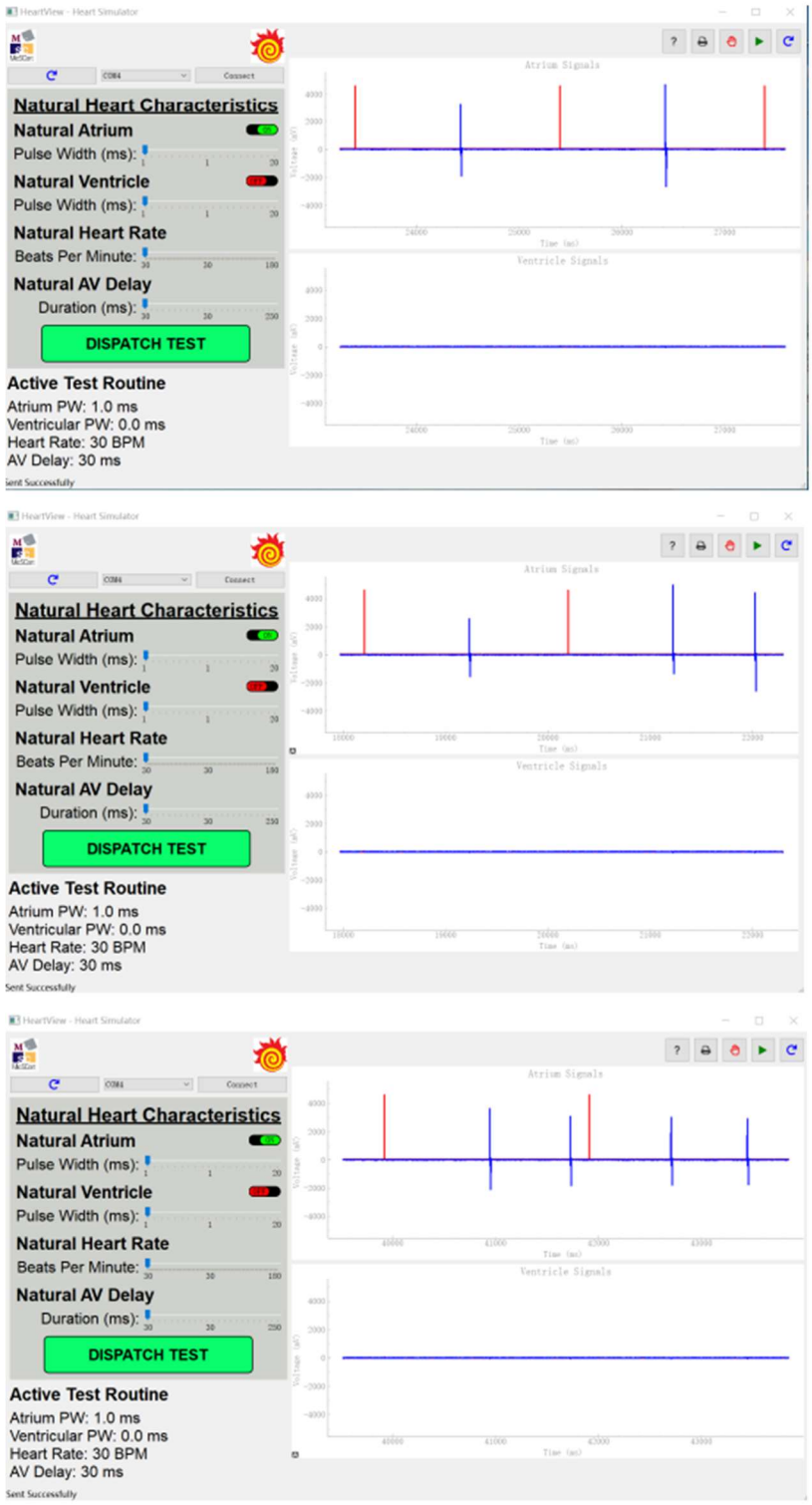


Figure 27: AAIR Test Case

The graphs depict AAIR mode. The first two plots show the pacemaker remaining stationary with the same behavior as in normal sensing mode. The third picture demonstrates an increase in pacing rate, with the pacemaker maintaining the interval after sensing a natural heartbeat when shaken.

Condition: Patient with natural heartbeat and increasing physical activity.

Expected Output: Pacemaker paces with atrial and pacing rate increases.

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

VVIR Test Cases:

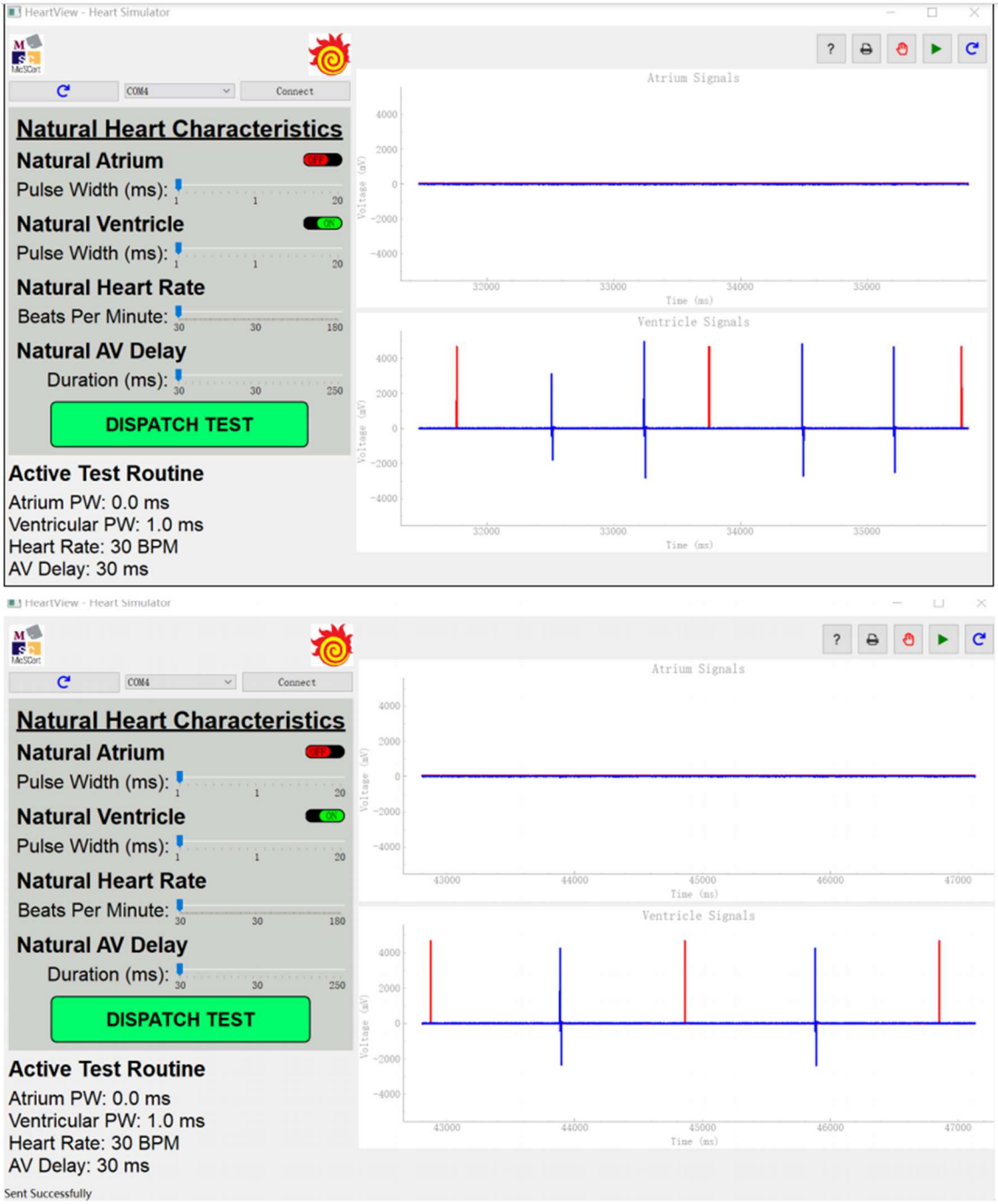


Figure 28: VVIR Test Case

In VVIR mode, the pacemaker exhibits a slower pacing rate when shaken, but maintains the required inhibition time interval after sensing a natural heartbeat while at rest.

Condition: Patient with natural heartbeat and decreasing physical activity.

Expected Output: Pacemaker paces with ventricle and pacing rate decreases.

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

DOO Test Cases:

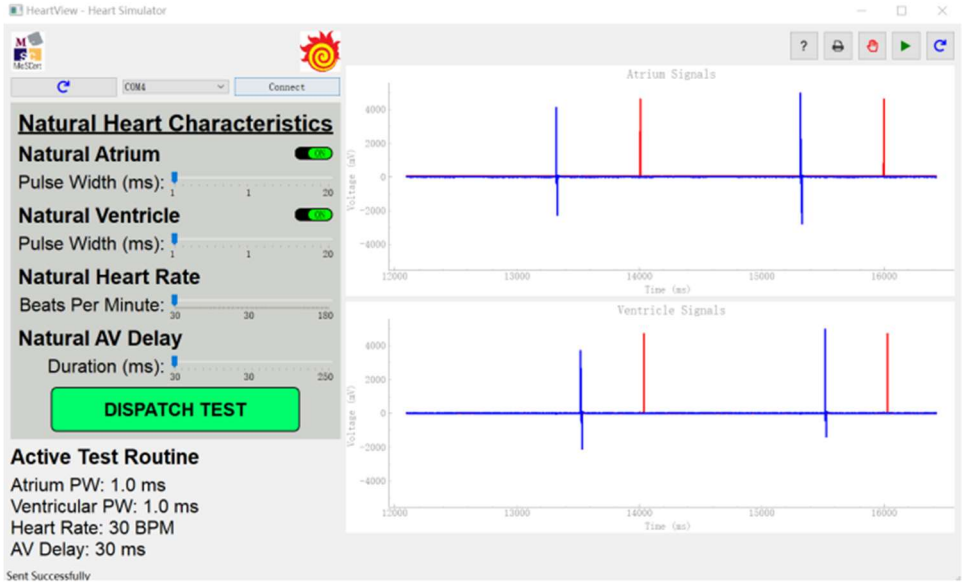


Figure 29: DOO Test Case

Condition: Patient with natural heartbeat.

Expected Output: Pacing order and time interval are not in accordance with the input setting.

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

DOOR Test Cases:

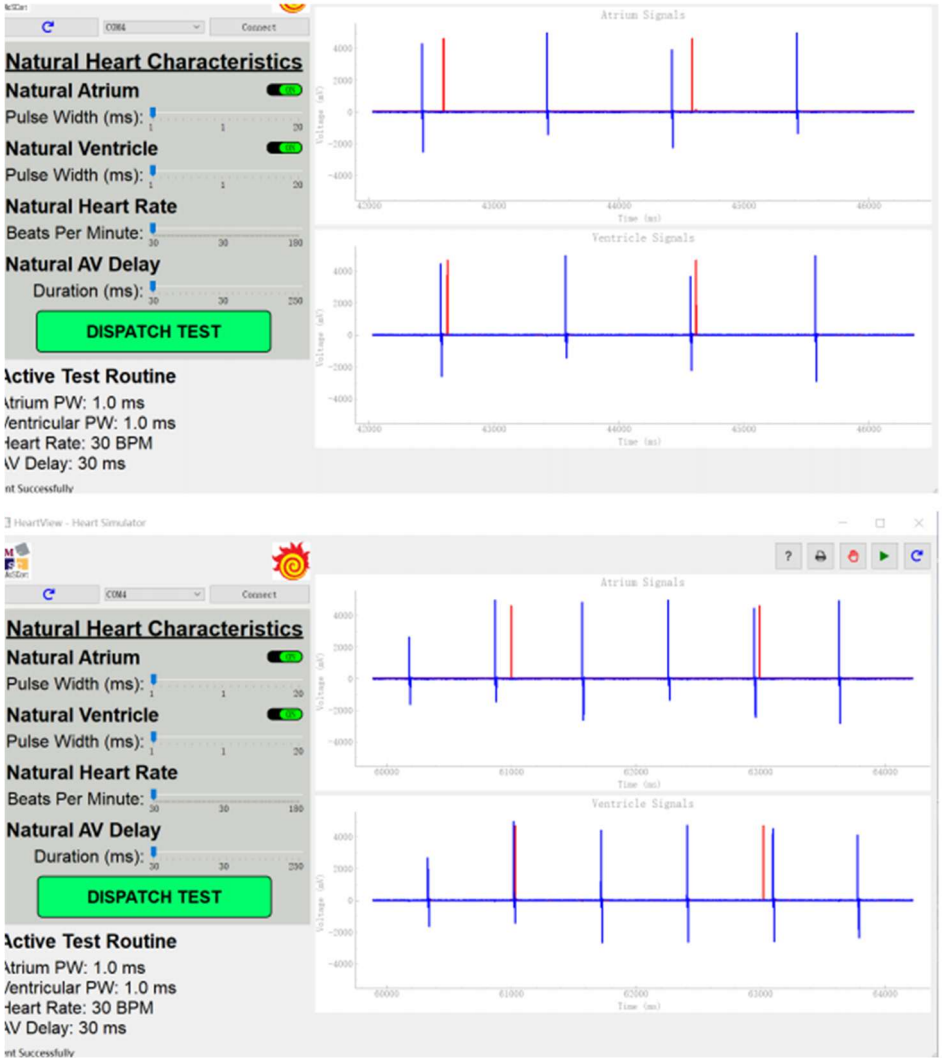


Figure 30: DOOR Test Case

The picture displays DOOR mode. The first graph illustrates the pacemaker placed stationary on the table, while the second graph shows an increase in pacing rate when the pacemaker is shaken.

Condition: Patient with natural heartbeat.

Expected Output: Pacemaker paces with the heart chambers and pacing rate increases.

Actual Output: Pacemaker paces as expected result

Pass/Fail: Pass

1.5 Potential Changes to Simulink

Assignment 1:

Given our initial modes, we can further update and change the Simulink design to encompass the potential new modes that will be added such as the AOOR, VOOR, AAIR, VVIR. We can also add the DDD and DDDR operating modes as a bonus change. Additionally, another potential change is that the rate adaptive modes must track activity using the on-board accelerometer. The pulse rate (LRL) should increase once sufficient activity is detected.

Assignment 2:

Currently, we are using 16bit memory to store the values, with further testing, we found out that this can waste a lot of memory and slow down the implementation in the long run. A potential change we can implement into the Simulink system would be to switch from 16bit to 8bit to store the values in. In the case the value does not fit in the 8 bit, we can decrement it (decrease the value), add another memory space and then revert back to the original value.

Part 2 – DCM Design

Device Controller-Monitor (DCM):

A DCM is a device used by a medical professional to monitor and alter the behavior of the pacemaker device. It allows one to provide instructions (programming) and receive information (telemetry) from the pacemaker device [2].

2.1 Current Requirements

The requirement of this DCM includes a graphical user interface (GUI) that allows the physician (user) to get data and/or change data from the programmable pacemaker device. With the user prompt, the GUI is required to transmit a new set of instructions to alter the behavior of the embedded software in the pacemaker device by modifying programmable parameters or sending instructions to interrogate the pulse generator.

2.1.1 Welcome/login Screen

The interactive welcome screen interface allows the physician (user) to register or login as a user into the platform. Here the user can store data linked to their user account. A max of 10 users can be stored locally.

2.1.2 User Interface

Firstly, The DCM user interface should be capable of utilizing and managing windows for display of text and graphics [1]. Secondly, it is required that the user interface should be capable of processing user positioning and input buttons with the addition of displaying all programmable parameters for review and modification. It should also be capable of visually indicating when the DCM and the device are communicating and visually indicate when telemetry is lost due to the device being out of range (or lost) [1]. Lastly, the user interface should be capable of visually indicating when a different Pacemaker device is approached than was previously interrogated by the Physician (user) [1].

2.1.3 DCM Utility Functions

This part of the interface provides standard information and features for the user to conveniently use the interface. The required DCM utility functions is as follows:

DCM Utility Functions Chart

Function Name	Features / Description
About	• Application Model Number • Software revision number • DCM Serial number • Institution name

Set Clock	• Set date • Set time
New Patient	• Interrogate new device without exiting the software application
Quit	• End a telemetry session

Table 2: DCM Utility Functions

2.1.4 Printed Reports

The user will be able to print out reports to get information on data such as pacemaker parameters, test results, histograms and more. Each report is required to contain the following header information: Institution name, Date and time of report printing, device model and serial number, DCM serial number, Application model and version number, report name.

The following printed reports are available for the physician (user):

Printed Reports:

Printed Report Name	Description
Parameter / Status reports:	
Bradycardia Parameters Report	-
Temporary Parameters Report	-
Implant Data Report	Storing lead implant date and polarity information. Stores pacing thresholds and P and R-wave amplitude functions
Threshold Test Report	Automatic pacing threshold test in AAI, VVI, and DDD modes of the DCM for both pulse width and amplitude measurements.
Measured Data Report	-
Marker Legend Report	-
Session Net Change Report	If parameter values are changed during the follow-up visit, the new setting is verified by viewing the "Session Net Change" report

Final Reports	This report will consist of the Measured Data, Threshold Test, Trending, Histograms, Implant Data, and Net Change reports.
Bradycardia Diagnostic reports:	
Rate Histogram Report	Involves the pacing rate and intrinsic rate distributions from a histogram recording period. Distributions shall be recorded for: paced atrial events, sensed atrial events, paced ventricular events, sensed ventricular events.
Trending Report	Involves pacing rate and sensor data
Electrogram (egram)	Available from the atrial and ventricular sense/pace leads and a surface electrogram. It is a visual output of the electrical activity inside each chamber of the heart. For our hardware, pins VENT SIGNAL and ATR SIGNAL are electrically connected to the analog egram signals for each chamber.

Table 3: User Available Printed Reports

2.1.5 Pacing Modes

The Bradycardia Operating modes are required in the DCM to be programmed. The operating/pacing modes implemented are: AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, VVIR.

AOO: The pacemaker consistently sends electrical pacing signals to the atrium without detecting the heart's natural beats or responding based on the heart's natural activity.

VOO: The pacemaker delivers electrical signals to the ventricle at a regular pace but does not actively detect the natural heart beats or respond to them.

AAI: The pacemaker sends pacing signals to the atrium, inhibits the response if sensing the natural atrial heart beats.

VVI: The pacemaker delivers pacing signals to the ventricle, monitors the natural heartbeats in the ventricle, and inhibits pacing response if sensing the natural heartbeat.

AOOR: The pacemaker sends electrical signals in the atrium at a regular pace without detecting natural heart beats. Furthermore, pacing rate alters based on the patient's physical activity.

VOOR: The pacemaker delivers electrical signals to simulate contractions in the ventricle and does not actively sense the natural heartbeat in the ventricle, but pacing rate can be adjusted based on the patient's physical activity.

AAIR: The pacemaker sends electrical signals at a regular pace in the atrium, monitors the natural atrial beats. If the pacemaker senses a natural heartbeat in the atrium, it inhibits pacing response, and its rate can be altered by the patient's physical activity.

VVIR: The pacemaker delivers electrical signals to the ventricle, senses the natural ventricle beats, and inhibits the response when detecting intrinsic ventricle activity. Additionally, the VVIR mode allows the pacing rate to be changed based on the levels of the patient's activity.

DOO: The pacemaker delivers regular pacing to the heart's chamber, including atrium or ventricle, without sensing the natural heartbeats.

DOOR: The pacemaker provides a fixed-rate pacing that is dependent on the patient's activity level to the heart's chamber.

2.1.6 Programmable and Measured Parameters

The user interface is required to display these programmable parameters. They are provided for controlling the delivery of patient-tailored bradycardia therapy. This chart includes the programmable parameters based on their nominal value, max and min programmable values, their increment and lastly their corresponding units.

Programmable and Measured Parameters in the interface

Parameter	Nominal Value	Min Value	Max Value	Increment	Units
Programmable Parameters					
Lower Rate Limit	60	30	175	5	ppm
Upper Rate limit	120	50	175	5	Ppm
Maximum Sensor Rate	120	50	175	5	Ppm
Fixed AV Delay	150	70	300	10	ms

V Pulse Amplitude	5.0	0	5	0.1	V
A Pulse Amplitude	3.5	0	5	0.1	V
V Pulse Width	1.0	0.1	1.9	0.1	ms
A Pulse Width	1.0	0.1	1.9	0.1	ms
V Sensitivity	1.0	1.0	10	0.5	mV
Ventricular Refractory Period (VRP)	320	150	500	10	ms
Atrial Refractory Period (ARP)	250	150	500	10	ms
PVARP	250	150	500	10	ms
Activity Threshold	Med	-	-	V-Low, Low, Med-Low, Med, Med-High, High, V-High	-
Reaction Time	30	10	50	10	Sec
Response Factor	8	1	16	1	-
Recovery Time	5	2	16	1	min
Measured Parameters					
P and R wave measurement	-	-	-	0.1	mV
Lead Impedance	-	-	-	50	Ω
Battery Voltage	-	-	-	0.01	V

Table 4: Programmable and Measured Parameters

2.1.7 Diagnostics and communication

The system will provide diagnostic tools such as: Measured data, Threshold Test, Rate training, Histogram and real-time data. P and R wave measurements will also be displayed when commanded by the DCM. The user also has the option of viewing real-time electrograms, either through the screen or from a printed copy. The user can access the following Internal electrogram (EGM) options: an atrial EGM, a ventricular EGM or both.

2.1.8 Serial Communication

In order to send programmable parameters, start the pacemaker's pacing, and get telemetry data from the pacemaker, the DCM implements serial communication with the device. The programmable parameters originate at the DCM and are implemented in the device.

The Simulink model configures an input block to store initial values for all parameters, including SET_PARAM and ECHO_PARAMETERS states. The first state receives serial input and assigns them to variables within the Simulink. Meanwhile, the second state is employed to transmit the resultant data to the DCM through the 'send_parameters' function. Furthermore, the ATR_SIGNAL and VENT_SIGNAL variables are mapped to A0 and A1 on the pin map, facilitating the transmission of atrial or ventricular signals back to the DCM. The designated data types provide sufficient precision for parameter values within an acceptable range, increasing processing speed and memory efficiency.

2.1.9 Potential changes to DCM Assignment 2 requirements

The requirements for the DCM design for this assignment required us to expand the DCM to include all the required modes and parameters. The parameters also needed to be updated to reflect the specifications of the system. This assignment required us to implement serial communication via UART to transmit and receive information between the DCM and the Pacemaker device. The system is also able to set, store and transmit these parameters using serial communication and the use of the UART COM ports. The parameters are stored within a unique .json file that is saved for each user, which can be opened at any time. After submitting the parameters, the DCM is able to plot the electrocardiogram and display it to the user depending on what kind of graph was requested (Ventricle, Atrium or Both).

The feedback given in Assignment1 Demo was also implemented in Assignment2.

1. Adding modularity to the design.
2. Defining constraints and limits for the programmable parameters
3. Adding an logout function in the design for the user

2.2 Design decisions

For Assignment 1, the choice to develop the Device Communication Module (DCM) GUI using Python and the tkinter library was made after careful analysis focused on the unique benefits and project-specific design specifications. Python was the best option for creating a user interface in a medical setting because of its readability and versatility. Cross-platform compatibility was critical because the DCM GUI needed to work flawlessly across Windows, macOS, and Linux, among other operating systems. A more simple and manageable design process was made possible by Python's clean syntax and simplicity, which is important in a medical application where accuracy and clarity are critical.

Python's broad library support was one of its main features, and tkinter, a standard GUI library, gave users access to a large collection of pre-made graphical elements including buttons and input fields. The lack of necessity to build these components from the ground up allowed for an acceleration of development thanks to this extensive toolset. Another strong argument in favour of Python's adoption was its lively and supportive community, which provides a plethora of information, documentation, and support from other members. In the highly regulated and standards-driven healthcare industry, these qualities proved to be invaluable in resolving issues and guaranteeing compliance with recommended procedures.

For the DCM GUI to communicate with external medical monitoring devices and backend systems, Python's smooth integration features were essential. Python's ability to seamlessly integrate with a wide range of technologies via libraries and APIs expedited communication across various medical system components. In the end, tkinter and Python were selected over alternative languages and libraries because they met the particular requirements of the project. Python is the safest option for a medical GUI because of its cross-platform compatibility, readability of code, speed of development, and strong community support—especially in a setting where accuracy, dependability, and standard compliance are crucial. This choice guaranteed the DCM GUI's effective and user-friendly design, matching it with the exacting standards of the healthcare industry.

For Assigment2, Current design decisions that are likely to change include:

1. Wrap the windows module with a class to increase the modularity and information hiding.
2. Create the class to support UART between pacemaker and Simulink.
3. Show the parameters review and modification on the same tab to increase readability.
4. create the logout function for the users.
5. Provide an interface to check the history of the parameter’s modification in the application.
6. Use the slider input for parameters to enforce the valid input.
7. Add choice for resetting the parameters.
8. Make better appearance for egram.

Assignment 1:

2.2.1 Maximum Number of Users Allowed

The first design decision was to limit the number of registered users. As mentioned in the requirements, the number of users is limited to 10.

Code:

```
# Maximum number of users allowed
MAX_USERS = 10
```

2.2.2 User Authentication

The second design decision was to authenticate users based on their username and password. As seen in the figure below, the login function should check if the user is already registered or not. The register function should allow the user to create their unique username and password.

Code:

```
# Function to handle user login
def login():
# Function to handle user registration
def register():
```

2.2.2 User Data Storage

The third design decision was to store user data in JSON files for the login and register function described above to work properly.

Code:

```
# Function to load user data from a JSON file
def load_users():


# Function to save user data to a JSON file
def save_users(data):
```

2.2.3 User Variable Storage

The fourth design decision was to store user-specific variables in JSON files, one for each mode. The variables will be stored individually for each user.

Code:

```
# Functions to load and save user variables for a specific mode
def load_user_variables(username, mode):

def save_user_variables(username, mode, variables):
```

2.2.4 Mode Selection:

The fifth design decision was to allow users to select a mode (AAI, VVI, VOO, AOO).

Code:

```
# Function to show the mode selection page
def show_mode_select(username):
```

2.2.6 User Variable Display:

The sixth design decision was to display user-specific variables based on the selected mode. After the user has logged in, there should be a page where the user is able to select modes and their corresponding parameters.

Code:

```
# Function to show the welcome page with user-specific variables
def show_welcome_page(username, mode):
```

2.2.7 Variable Update:

The seventh design decision was to allow users to update their variables and save those updates after the function is executed.

Code:

```
def update_variables():
```

2.2.8 Main User Interface:

The eighth design decision was to create a graphical user interface (GUI) for user login and registration.

Code:

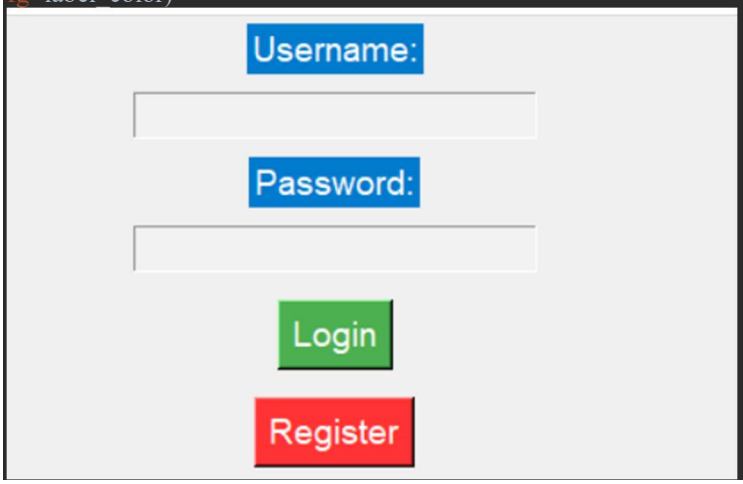
```
# Create the main login/registration window
login_window = tk.Tk()
```

2.2.9 GUI Elements:

The ninth design decision was to create labels, entry widgets, and buttons for user interaction.

Code:

```
username_label = tk.Label(login_window, text="Username:", font=font, fg=label_color, bg="#007acc")

username_entry = tk.Entry(login_window, font=font, bg=entry_bg_color)

password_label = tk.Label(login_window, text="Password:", font=font, fg=label_color, bg="#007acc")

password_entry = tk.Entry(login_window, show="*", font=font, bg=entry_bg_color)

login_button = tk.Button(login_window, text="Login", command=login, font=font, bg="#4CAF50",
fg=label_color)

register_button = tk.Button(login_window, text="Register", command=register, font=font, bg="#ff3333",
fg=label_color)
```



2.2.10 DCM Connection Status Display:

The 10th design decision was to display DCM connection status.

Code:

```
dcm_label = tk.Label(login_window, text="DCM connected", font=font, fg="green")
```

2.2.11 Pacemaker Detection Display:

The 11th design decision was to display the ID of the pacemaker currently being used.

Code:

```
PACEMAKER_label = tk.Label(login_window, text="PACEMAKER ID:1 is currently being used",
font=font, fg="green")
PACEMAKER_label.pack()
```

2.2.12 Egram data Display:

The 12th design decision was to display the Egram data tab of the pacemaker currently being used.

- For this assignment, this only works as a empty tab, for future assignments, this will be properly implemented.

Code:

```
# Function for the "Egrams" button (you can customize this function's behavior)
def egrams_function():
    messagebox.showinfo("Egrams", "This is the Egrams feature. You can customize its behavior here.")
```

2.3 Advanced Design Decisions:

The advanced design decisions for this medical system or pacemaker control application encompass a range of enhancements aimed at improving system functionality and user experience. First and foremost, a maximum limit of registered users, set at 10 in the code (MAX_USERS = 10), helps control system usage, preventing overuse. User data, such as usernames and passwords, is now stored in JSON files for persistent data storage. Functions have been implemented for loading and saving user data to and from these files. In addition, user-specific variables for each operational mode (AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR, VVIR) are saved in separate JSON files. Functions enable the loading and saving of these variables, promoting a more personalized user experience. User authentication is another significant feature, requiring users to log in with their username and password. If credentials match stored data, access is granted; otherwise, a login failure message is displayed.

To facilitate user interaction and mode selection, a graphical user interface (GUI) has been introduced. This GUI includes labels, entry widgets, and buttons for user login and registration, enhancing the overall user experience. Furthermore, a DCM (Device Communication Module) connection status display informs users about the system's connection to external devices. When the DCM is connected, a green "DCM connected" label is presented to provide this critical feedback. These advanced design choices collectively make the medical system more robust, user-friendly, and equipped with the necessary tools for data management and pacemaker control while ensuring user security and personalization.

2.3.1 Maximum Number of Users Allowed

This decision was made to limit the number of registered users to prevent overuse of the system.

Code:

```
# Maximum number of users allowed
MAX_USERS = 10
```

2.3.2 User Data Storage

This advancement was made to store user data, including usernames and passwords, in JSON files for persistent data storage.

Code:

```
# Function to load user data from a JSON file
def load_users():
    try:
        with open("users.json", "r") as file:
            return json.load(file)
    except FileNotFoundError:
```

```
        return {}

# Function to save user data to a JSON file
def save_users(data):
    with open("users.json", "w") as file:
        json.dump(data, file)
```

2.3.3 User Variable Storage

This decision was made to store user-specific variables for each mode in separate JSON files.

Code:

```
# Function to load user variables from a JSON file for a specific mode
def load_user_variables(username, mode):
    try:
        with open(username + f"_{mode}_variables.json", "r") as file:
            return json.load(file)
    except FileNotFoundError:
        return {}

# Function to save user variables to a JSON file for a specific mode
def save_user_variables(username, mode, variables):
    # Load existing user variables for the selected mode
    existing_variables = load_user_variables(username, mode)

    # Update existing variables with new values
    existing_variables.update(variables)

    # Save user variables for the selected mode back to the JSON file
    with open(username + f"_{mode}_variables.json", "w") as file:
        json.dump(existing_variables, file)
```

2.3.4 User Authentication

This decision was made to authenticate users based on their username and password.

Code:

```
# Function to handle user login
def login():
    global current_user
    global dcm_connected  # Declare the global variable

    username = username_entry.get()
    password = password_entry.get()

    if username in users and users[username] == password:
        current_user = username  # Set the current user
        dcm_connected = True  # Set DCM connection status to True
        show_mode_select(username)
    else:
        messagebox.showerror("Login Failed", "Invalid username or password")
```

2.3.5 Mode Selection

This decision was made allow users to select one of four modes (AAI, VVI, VOO, AOO).

Code:

```
# Function to show the mode selection page
def show_mode_select(username):
    # Destroy the current window
    if 'welcome_window' in globals():
        welcome_window.destroy()

    # Create the mode selection window
    mode_window = tk.Tk()
    mode_window.title("Mode Selection")
    mode_window.geometry("400x250")

    # Create labels and buttons for mode selection
    label = tk.Label(mode_window, text="Select Mode", font=("Arial", 14))
    label.pack(pady=10)
```

2.3.6 User Variable Display

This decision was made to display user-specific variables based on the selected mode.

Code:

```
# Function to show the welcome page with user-specific variables
def show_welcome_page(username, mode):
    # Load user variables from JSON file for the selected mode
    user_variables = load_user_variables(username, mode)

    # Create the welcome page window
    global welcome_window
    welcome_window = tk.Tk()
    welcome_window.title("Pacemaker Control Panel")
    welcome_window.geometry("800x400")

    # Create Labels for variables with professional font and colors
    font = ("Arial", 14)
    label_color = "#333"
```

2.3.7 Variable Update

This decision was made to allow users to update variables and save those updates.

Code:

```
def update_variables():
    updated_variables = {}
    for var_name, entry in variable_entries.items():
        updated_value = entry.get()
        updated_variables[var_name] = updated_value
```

```
    # Save all updated user variables for the selected mode
    save_user_variables(username, mode, updated_variables)

    messagebox.showinfo("Variables Updated", "User variables updated successfully.")
```

2.3.8 Main User Interface

This decision was made create a graphical user interface (GUI) for user login and registration.

Code:

```
# Create the main login/registration window
login_window = tk.Tk()
login_window.title("Medical System")
login_window.geometry("400x250")
```

2.3.9 GUI Elements

This decision was made to create labels, entry widgets, and buttons for user interaction.

Code:

```
3       username_label = tk.Label(login_window, text="Username:", font=font, fg=label_color, bg="#007acc")
        username_label.pack(pady=5)
        username_entry = tk.Entry(login_window, font=font, bg=entry_bg_color)
        username_entry.pack(pady=5)

        password_label = tk.Label(login_window, text="Password:", font=font, fg=label_color, bg="#007acc")
        password_label.pack(pady=5)
        password_entry = tk.Entry(login_window, show="*", font=font, bg=entry_bg_color)
        password_entry.pack(pady=5)

        login_button = tk.Button(login_window, text="Login", command=login, font=font, bg="#4CAF50",
        fg=label_color)
        login_button.pack(pady=10)

        register_button = tk.Button(login_window, text="Register", command=register, font=font, bg="#ff3333",
        fg=label_color)
        register_button.pack(pady=5)
```

2.3.10 DCM Connection Status Display

This decision was made to display the DCM connection status to inform the user.

Code:

```
# Display "DCM connected" message if DCM is connected
if 1:
    dcm_label = tk.Label(login_window, text="DCM connected", font=font, fg="green")
    dcm_label.pack()
```

These detailed design decisions and corresponding code parts provide a comprehensive overview of the program's structure and functionality, ensuring user authentication, mode-specific variable handling, and a user-friendly graphical interface.

Module 1: User Data Management

The purpose of this module manages the loading and saving of user data from a JSON file. It serves as the backend for user authentication and registration processes.

The public functions are as follows:

1. load_users(): Loads user data from the "users.json" file.
    - Parameters: None
2. save_users(data): Saves user data to the "users.json" file.
    - Parameters: data (dictionary containing user data)

The functions with Black-box Behavior are as follows:

1. load_users(): Reads the "users.json" file and returns user data as a dictionary. If the file doesn't exist, it returns an empty dictionary.
2. save_users(data): Writes the provided data to the "users.json" file.

The state variables implemented are:

1. users (dictionary) - Stores user data.

There were no Private Functions included.

The Internal Behavior of the functions:

1. load_users():
    - Internally opens and reads the "users.json" file.
    - Parses the file's contents as JSON and returns it as a dictionary.
    - If the file doesn't exist, returns an empty dictionary.
2. save_users(data):
    - Internally opens the "users.json" file in write mode.
    - Writes the provided data (a dictionary) as a JSON-formatted string to the file.

Module 2: User Variables Management

The purpose of this module handles loading and saving user-specific variables for each mode (AAI, VVI, VOO, AOO) using JSON files. It manages user-specific settings for different pacemaker modes.

The public functions included in this module is as follows:

1. load_user_variables(username, mode): Loads user-specific variables for a specific mode.
    - Parameters: username (string), mode (string)
2. save_user_variables(username, mode, variables): Saves user-specific variables for a specific mode.
    - Parameters: username (string), mode (string), variables (dictionary of user-specific variables)

The Black-box behavior functions in the module:

1. load_user_variables(username, mode): Reads the <username>_<mode>_variables.json file and returns user variables as a dictionary. If the file doesn't exist, it returns an empty dictionary.
2. save_user_variables(username, mode, variables): Writes the provided variables dictionary to the <username>_<mode>_variables.json file.

There are no state variables or private functions in this module:

The Internal behavior in this module is as follows:

1. load_user_variables(username, mode):
    - Internally opens and reads the <username>_<mode>_variables.json file.
    - Parses the file's contents as JSON and returns it as a dictionary.
    - If the file doesn't exist, returns an empty dictionary.
2. save_user_variables(username, mode, variables):
    - Internally opens the <username>_<mode>_variables.json file in write mode.
    - Writes the provided variables (a dictionary) as a JSON-formatted string to the file.

Module 3: User Authentication

The purpose of this module authenticates users based on their provided username and password. It handles user login and registration processes.

The public functions included in this module:

1. login(): Handles user login.
    - Parameters: None
2. register(): Handles user registration.
    - Parameters: None

The black-box behavior is as follows:

1. login():
    - Validates the provided username and password against stored user data.
    - If successful, sets current_user to the username and dcm_connected to True.
    - Displays an error message if login fails.
2. register():
    - Checks if the maximum number of users is reached and if the provided username already exists.
    - Creates a new user account if conditions are met, including setting current_user to the new username.
    - Displays an error message if registration fails.

The state variables are as follows:

- users (dictionary) - Stores user data.
- current_user (string) - Tracks the current user.
- dcm_connected (boolean) - Indicates DCM connection status.

There are no private functions for this module.

The internal behavior of the functions includes:

1. login():
    - Internally compares the provided username and password with the stored user data.
    - If the credentials match, it sets current_user and dcm_connected.
    - If login fails, it displays an error message using the messagebox.

2. register():
    - Checks the user limit and existing usernames to determine if registration is allowed.
    - If conditions are met, it creates a new user account by updating the user's dictionary.
    - It sets current_user to the newly registered username.
    - If registration fails, it displays an error message using the messagebox.

Module 4: Mode Selection

The purpose of this module handles the mode selection process, allowing users to choose between AAI, VVI, VOO, and AOO modes. It initiates the selected mode and moves to the next step.

The public functions of this module include:

1. show_mode_select(username): Displays the mode selection window.
    - Parameters: username (string)

The Black-box behavior of this module is as follows:

1. show_mode_select(username):
    - Displays a window with mode selection buttons (AAI, VVI, VOO, AOO).
    - Selecting a mode initializes user variables for that mode and moves to the welcome page.

There are no state or private variables for this module.

The internal behavior of this module is as follows:

1. show_mode_select(username):
    - Creates a window with mode selection buttons.
    - Initializes user variables for the selected mode (e.g., save_user_variables(username, mode, {})).
    - Proceeds to the welcome page when a mode is selected.

Module 5: User Variable Display and Update

The purpose of this module displays user-specific variables based on the selected mode and allows users to update these variables. It provides an interface for variable modification and saving changes.

The public functions are as follows:

1. show_welcome_page(username, mode): Displays the welcome page with variable entry fields.
    - Parameters: username (string), mode (string)

The module contains these black-box behaviors:

1. show_welcome_page(username, mode):
    - Displays a window with labels and entry fields for user variables.
    - Allows users to modify variables and save changes.

The state variables of this module is as follows:

- welcome_window - Reference to the welcome page window.
- variable_entries (dictionary) - Tracks user variable entry fields.

The private functions are as follows:

1. update_variables(): Handles variable updates and saves changes.
    - Parameters: None

The Internal behaviors of this module:

1. show_welcome_page(username, mode):
    - Loads user-specific variables for the selected mode from JSON files.
    - Creates a window with labels and entry fields for each variable.
    - Initializes entry fields with existing values.
    - Allows users to modify variable values.
    - Calls update_variables() to save changes and displays a confirmation message.
2. update_variables():
    - Iterates through variable entry fields and retrieves updated values.
    - Saves the updated variables for the selected mode using save_user_variables(username, mode, updated_variables).
    - Displays a confirmation message using the messagebox.

Module 6: Main Application

The purpose of this module serves as the entry point for the program, creating the main login/registration window and handling user interactions for authentication and mode selection.

There are no public functions for this module.

The black-box behavior in this module:

- Displays the main login/registration window.
- Handles user login, registration, and displays DCM connection status.

There are no state variables or private fucntions in this module.

The Internal behavior of this module:

- Creates the main login/registration window with labels, entry widgets, and buttons.
- Handles user login and registration using login() and register() functions.
- Displays the "DCM connected" message if the DCM is connected.

These detailed descriptions cover the purpose, public functions, black-box behaviors, state variables, private functions, and internal behaviors for each module in the program.

Assignment 2:

2.3.1 Class Windows

The windows.py module contains the parameters for the GUI. Here, the visual windows can be modified directly with no effect on the decision-making process for the transitions between interfaces. Only the visual portion is contained within this module. The windows are generated using the Python package PySimpleGUI, which was chosen for its visual simplicity and ease of use. Each window is stored as a separate function and can be individually edited. The functions are as follows:

Public Functions:

1. create_welcome_window()

This function uses PySimpleGUI to format the visual appearance of the GUI of the welcome window where the user selects to login or register. It also accepts user inputted information and passes it through to the main module for data processing and allows for the decision flow to the next window. Note that there are no input parameters as the function is purely visual.

2. create_login_window()

This function formats the visual appearance of the login window. The user is prompted to enter their username and password. No input parameters are needed.

3. create_register_window()

This function formats the visual appearance of the register window. The user is prompted to create a username and password. No input parameters are needed.

4. create_main_window()

This function formats the visual appearance of the main window. The pacing mode, lower rate limit, atrial amplitude, atrial pulse width, ARP, upper rate limit, ventricular amplitude, ventricular pulse width, VRP, AVDelay, activity threshold, reaction time, response factor, recovery time are all listed. A separate tab is included to allow the user to modify the values. The buttons for communicating with the pacemaker are also included. No input parameters are needed.

5. create_egram_window()

This function formats the window showing the egram.

2.3.2 Class Users

The users.py module only contains the back-end logic design that stores and modifies the user information including the usernames and passwords of up to ten unique users, as well as the function for logging in and registering new users. There is no modification of the GUI (visual or interactive) in this module. The functions in this module are as follows, all belonging to a public class Users:

Public Functions:

1. **get_active(self)**

This function returns the latest list of active users.

2. **login(self, values)**

This function attempts to check the user inputted login information with an existing user already stored. It will output true on success, and false otherwise.

3. **register(self, values)**

This function takes the user inputted registration information and attempts to register a new user. If the inputted username is already in use, the inputted entries are empty, or the maximum number of users have been registered, it will return false.

Global Variables:

1. **USERS_FILE**

This variable stores the directory path of the local file storing the user information

2. **DEFAULT_USER**

This variable is the default record of a single user. The default username and password are both “admin”.

3. **USER_KEYS_TO_ELEMENT_KEYS**

This variable uses a dictionary mapping user keys to the window’s element keys. It maps the username to “-USERNAME-” and password to “-PASSWORD-”

4. **self.users_active**

This variable is a blank array that is used to store the current active users.

5. **self.info**

This variable holds all the users records in a list of dictionaries. The runtime equivalent of local file.

Private Functions:

1**. __new__(cls)**

This is the constructor for the Users object. It outputs an initialized instance of the Users class.

2. **load_users(self, users_file, default_user)**

This function takes in the file path where the user records are stored and loads the history from the file to self.info. In the event that no user information is found, a blank user file will be created using the default user admin.

3. **save_users(self, users_file)**

This function saves the current user information into the local file.

Internal Behavior:

1. **__new__(cls):**

This function first checks if there exists an instance of the Users class. If not, it then creates and initializes

the instance by calling the load_users function to get access to the stored user information.

2. **load_users(self, users_file, default_user)**

This function tries to load the local .json file storing the usernames and passwords in order to check the

user inputted values with the previously registered users. If no local file is found, it will create a blank file.

and automatically store a default user “admin”

3. **save_users(self, users_file)**

This function writes the user inputted information into the local .json file.

4. **get_active(self)**

This function returns the array of active users.

5. **login(self, values)**

This function checks the user inputted information to the previously loaded record of usernames and passwords. If they match, it will output true to bring the user to the next window through the main module. Otherwise, it will output false and prompt the user to try again.

6. **register(self, values)**

This function attempts to register a new user using the user inputted registration information. This will output logic false if no more users may be stored, if the entries are empty, or if the desired

username is already in use.

2.3.3 Class Pacemakers

This module is used to access and manipulate the history of modification of parameters. This module is implemented using the singleton pattern to ensure there is only one instance of the whole history. The history records are stored in a json file locally in the form of a list of dictionaries (in Python's terminology). It also handles the communication between DCM and the pacemaker to get and set parameters on the pacemaker in real-time.

Public Functions:

1. **get_param(self)**

This function returns the latest version of parameters.

2. **set_param(self, values)**

This function takes the input values of pacemaker parameters and updates the state of the history (both runtime state and local file).

3. **get_param_com(self)**

This function requests current parameters of the pacemaker and returns it in a dictionary for the main module to update the status tab.

4. **set_param_com(self)**

This function sends the latest saved parameters to the connected pacemaker.

Global Variables:

1. **MODE**

This variable stores the implemented modes of the pacemakers in a tuple. At this time, there are ten modes: "AOO", "AOOR", "VOO", "VOOR", "DOO", "DOOR", "AAI", "AAIR", "VVI", "VVIR".

2. **DEFAULT_PARAM**

This variable is the default record of the pacemaker parameters in a dictionary. It maps each parameter to None.

3. **PARAM_KEYS_TO_ELE_KEYS**

This variable is a dictionary mapping the parameters to elements in the status tab of the main window of the application. The key is a parameter, and the value is the element key (identifier for elements on the window) corresponding to specific parameters.

4. **PARAM_KEYS_TO_ELE_KEYS_IN**

This variable is a dictionary mapping the parameters to elements in the modify tab of the main window of the application. The key is a parameter, and the value is the element key (identifier for elements on the window) corresponding to specific parameters.

5**. MODE _TO_ELE _KEYS**

This variable is a dictionary mapping the modes of the pacemaker to the elements in the review tab corresponding to the parameters needed in each specific mode. The key is mode, and the value is a tuple of element keys.

6. **MODE_TO_PARAM _KEYS**

This variable is a dictionary mapping the modes of the pacemaker to the parameters needed in each specific mode. The key is mode, and the value is a tuple of parameters.

7. **PACEMAKERS_FILE**

This is the general path for the json file storing the history records.

8. **PORT**

This is the port name for serial communication,

9. **RATE**

This is the baud rate for serial communication, it is set to 115200 in this case.

10. **COM_ORDER**

This is the parameters order transferred between the DCM and the pacemaker.

11. **COM_FORMAT**

This is the format string used by struct to unpack the received byte stream.

12. **COM_PARAM_SIZE**

This is the byte number that needs to be read from the serial port when receiving data.

13. **COM_GET_H**

This is the header to initiate the process on the pacemaker to send the current parameters on it back to DCM.

14. **COM_SET_H**

This is the header to initiate the process on the pacemaker to set the parameters on it using the parameters sent by DCM.

15. **self.pacemakers_history**

This variable stores the history of modification in a list of dictionaries.

16. **self.pacemakers_current**

This variable stores the latest record of modification in a dictionary.

Private Functions:

1. **validate_keys(self, key, value)**

This function is used to validate the input for each parameter according to pacemaker specification. It takes in the key (parameter) and the value (user input), and returns True or False for validation.

2. **type_converter(self, key, value)**

This function converts the parameters type from string to specified type complying with that of pacemaker.

3. **__new__(cls)**

This is the constructor for the Pacemakers object. It outputs an initialized instance of the Pacemakers class.

4**. load_pacemakers(self, pacemakers_file)**

This function takes in the file path where the modification history records are stored and loads the history from the file to self.pacemaker_history. In addition, update the self.pacemaker_current to the latest state of the pacemaker.

4. **save_pacemakers(self, pacemakers_file)**

This function saves the modified history into the local file.

Internal Behavior:

1. **__new__(cls):**

This function first checks if there exists an instance of the Pacemakers. If not, it then creates and initializes the instance by calling the load_pacemakers function to get access to the history.

2. **load_pacemakers(self, pacemakers_file)**

This function tries to open the file storing the history and uses the json module to get the history into an object variable called pacemaker_history which is the runtime equivalent of the local file. And then it updates the variable storing the latest history record. If the file does not exist yet, it will create a file and add the DEFAULT_PARAM into it as well as updating the history and latest record variables.

3. **save_pacemakers(self)**

This function stores the content in the pacemakers_history back to the local file using the json module.

4. **get_param(self)**

This function constructs a dictionary mapping the element keys on the status tab to the latest parameters.

5. **set_param(self, values)**

This function constructs a dictionary mapping the parameters to the user inputs and updates the history and latest record. It first extracts the pacing mode parameter and uses the mode to determine which parameters are needed and then add them to the new dictionary after validating the inputs. The parameters not needed in specified pacing mode will get None as the value.

6. **validate_keys(self, key, value)**

This function uses multiple if statements to check the input for each parameter. The standard for

validating an input value is from the pacemaker specification.10

7. **type_converter(self, key, value)**

This function is similar to validate_key, but it assumes the keys have been validated which means it just needs to convert keys directly to desired types.

8. **get_param_com(self)**

This function sends header code packed by struct module to serial port using pyserial module which requests the pacemaker to send back the current parameters. Then it parses the bytes using the struct module to correct values. At last, it maps the parameter keys to window element keys for updating the status tab.

9**. set_param_com(self)**

This function does opposite operations to the get_param_com. It wraps the current parameters to bytes following the specified order in COM_ORDER and sends them to the serial port.

2.3.4 Class Egram

Public Functions:

1. draw_graph(self)

This function draws the egram on the area specified by the egram window.

Global Variables:

1. self.atr

This is a list holding the data point for signals from the atrium.

2. self.vent

This is a list holding the data point for signals from the ventricles.

3. self.fig

A figure object for egram to show.

4. self.ax

An ax object for detailed formatting of the egram.

5. self.canvas_agg

A handle from matplotlib to connect the figure object with the canvas object in the window.

6. self.mode

Mode of egram (atrial, ventricle, both).

Private Functions:

1. update_data(self)

This function updates the data point for generating a real-time egram.

2. clip(slef)

This function clips the data point to desired length for drawing.

Internal Behavior:

1. Update_data(self)

This function requests pacemaker in the same way as communication in pacemakers module to send back the signals from atrium and ventricles and add them to the self.atr and self.vent to update the graph in real-time.

2. clip(self)

This function handles the start of the egram where data points are not enough to fill the graph. It fills the extra empty point with 0.

3. draw_graph(self)

This function first updates the data points then draws the graph according to the self.mode, either drawing one of the atrium and ventricles or both. It calls the handle from the matplotlib.

4. **__init__(self, canvas, mode)**

This function is not a trivial initializer as it combines the canvas object of the window, the mode of the egram, together with the figure object.

2.3.5 Demo scenario for Rate Adaptivity:

The demo involves the pacemaker to be shaken by hand to simulate various speeds of activity; walking, jogging, and running. The device's rate should change accordingly. This was implemented through the egram graph. When the pacemaker was shaken in various speeds, the graph reacts and the rate changes. This change in movement can be seen in the egram graph.

2.4 Assurance Case

As we have designed a safety critical embedded system, the pacemaker, it is of critical importance that we need to generate assurance in why the system is safe [3]. This is referred to as an assurance case. The following sections are assurance cases for the safety requirements and specifications that we have implemented in the design.

DCM:

During the data transmission between DCM and Pacemaker, we make an encryption function to select the only data we need for transmission and it is then decoded using decoding function. It ensures the data safety.

We also created an assurance function in the register function which makes the users double check their created passwords in case they mistyped it in the first place. By doing this, the users are assured to log into the system.

Finally, we created the exit/logout function in every interface so that the user can choose to logout whenever they want to. By having a clear button indicating logout, the user will not be confused on how to sign out which if they are confused, they might start to press any other buttons which might lead to negative consequences.

Simulink:

We implemented hardware pin hiding in the Simulink design. Every submodule is embedded in the main module so that all the pin usage and parameters are kept away from user for safety assurance.

Assignment 1 Test Cases:

General Testing cases

Test Case 1: User Registration

Test Steps:

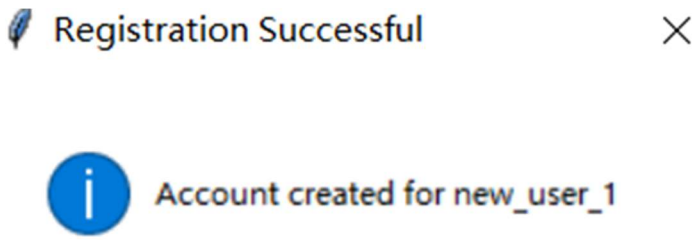
1. Start the application.
2. Click the "Register" button.
3. Enter a unique username (e.g., "new_user_1") and a password (e.g., "password123").
4. Click the "Register" button.

Expected Results:

- A successful registration message should appear.
- The user should be able to log in using the newly registered credentials.

Actual Results:

- A message indicates successful registration appeared on the screen.



Test Case 2: Duplicate Username Registration

Test Steps:

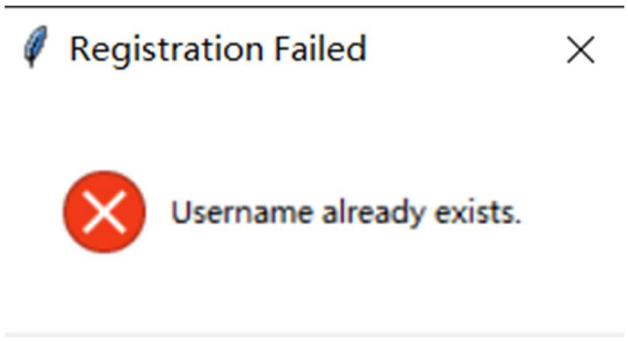
1. Start the application.
2. Click the "Register" button.
3. Enter an existing username (e.g., "user1") and a password (e.g., "password123").
4. Click the "Register" button.

Expected Results:

- An error message should appear, indicating that the username already exists.
- The user should not be registered with the same username.

Actual results:

- The error message appears on the screen.



Test Case 3: Blank Username or Password

Test Steps:

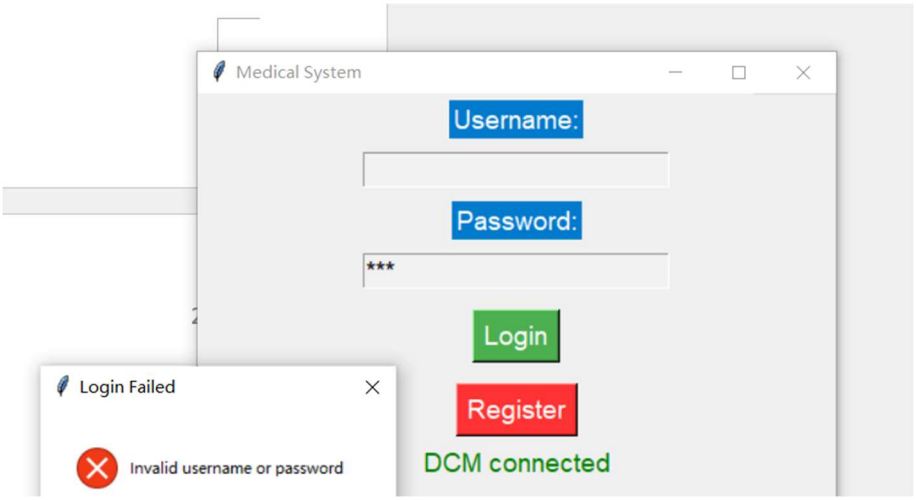
1. Start the application.
2. Click the "Register" button.
3. Leave the username field blank and enter a password (e.g., "password123").
4. Click the "Register" button.

Expected Results:

- An error message should appear, indicating that the username and password are required.
- The user should not be registered with blank fields.

Actual results:

- An error message appeared.



Test Case 4: Invalid Login

Test Steps:

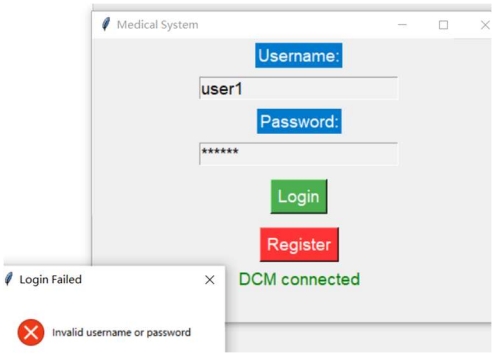
1. Start the application.
2. Click the "Login" button.
3. Enter a valid username (e.g., "user1") and an incorrect password (e.g., "wrong_password").
4. Click the "Login" button.

Expected Results:

- An error message should appear, indicating that the login failed.
- The user should not be logged in with an incorrect password.

Actual results:

- An error message appeared.



Test Case 5: Mode Selection

Test Steps:

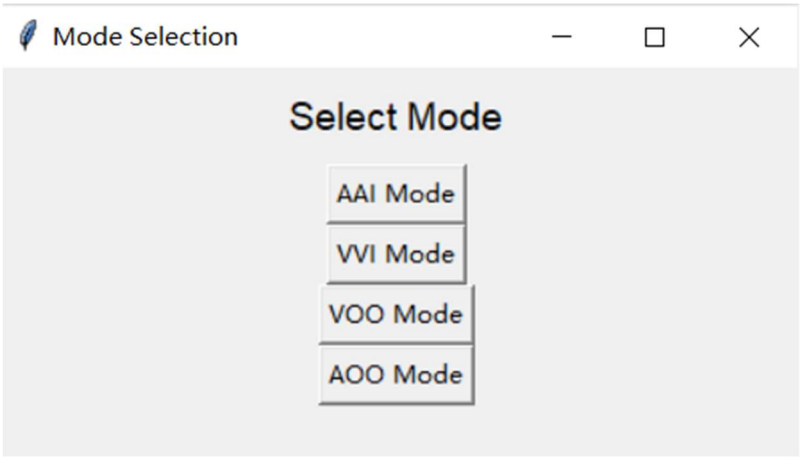
1. Start the application.
2. Register a new user (e.g., "new_user_2").
3. Select the "AAI" mode.
4. Click the "Back to Mode Selection" button.
5. Select the "VVI" mode.
6. Click the "Back to Mode Selection" button.
7. Select the "VOO" mode.
8. Click the "Back to Mode Selection" button.
9. Select the "AOO" mode.

Expected Results:

- The user should be able to switch between modes and return to the mode selection screen without issues.

Actual results:

- The program allows the users to switch between 4 operating modes.



Test Case 6: Variable Update Flow

Test Steps:

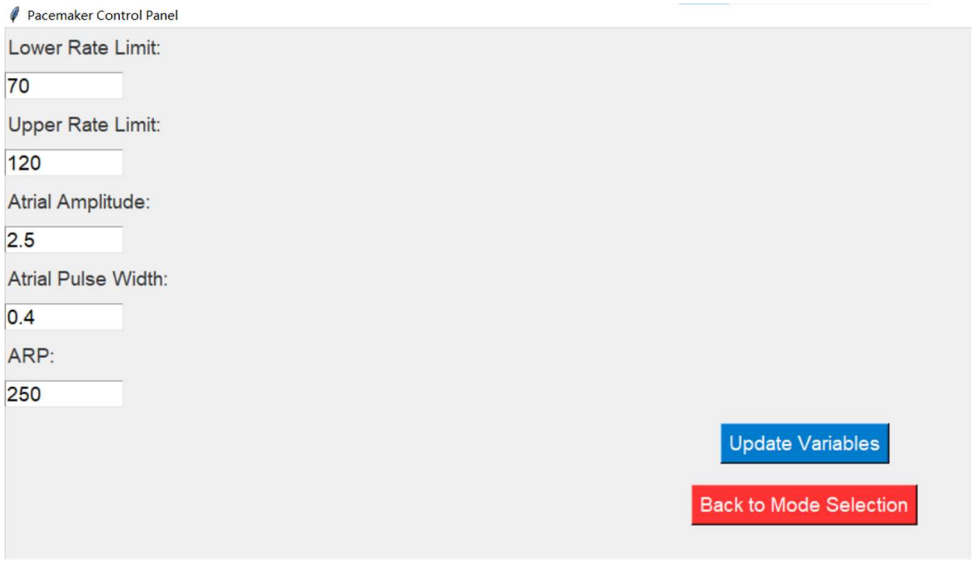
1. Start the application.
2. Register a new user (e.g., "new_user_3") and select the "AAI" mode.
3. Update variable values:
    - Lower Rate Limit: 70
    - Upper Rate Limit: 120
    - Atrial Amplitude: 2.5
    - Atrial Pulse Width: 0.4
    - ARP: 250
4. Click the "Update Variables" button.
5. Log out.
6. Log in with the same user ("new_user_3") and select the "AAI" mode.

Expected Results:

- Variable updates should be preserved between sessions.
- Logging in with the same user and mode should display the updated values (e.g., Lower Rate Limit: 70, Upper Rate Limit: 120, Atrial Amplitude: 2.5, Atrial Pulse Width: 0.4, ARP: 250).

Actual results:

- The updated variables along with its values are shown:



Test Case 7: Mode-Specific Variables

Test Steps:

1. Start the application.
2. Register a new user (e.g., "new_user_4") and select the "AAI" mode.
3. Update variable values:
    - Lower Rate Limit: 70
    - Upper Rate Limit: 120
    - Atrial Amplitude: 2.5
    - Atrial Pulse Width: 0.4
    - ARP: 250
4. Click the "Update Variables" button.
5. Select the "VVI" mode and update variables specific to VVI mode:
    - Lower Rate Limit: 60
    - Upper Rate Limit: 130
    - Ventricular Amplitude: 3.0
6. Click the "Update Variables" button.
7. Log out.
8. Log in with the same user ("new_user_4") and select the "AAI" mode.

Expected Results:

- Variable updates should be mode-specific, and changing variables in one mode should not affect variables in another mode.
- Logging in with the same user in a different mode should display the mode-specific variables.

Actual results:

- The variables are updated according to its operating mode, even thought they have the same variable name, their values are independent from each other.

Test Case 8: Invalid Variable Values

Test Steps:

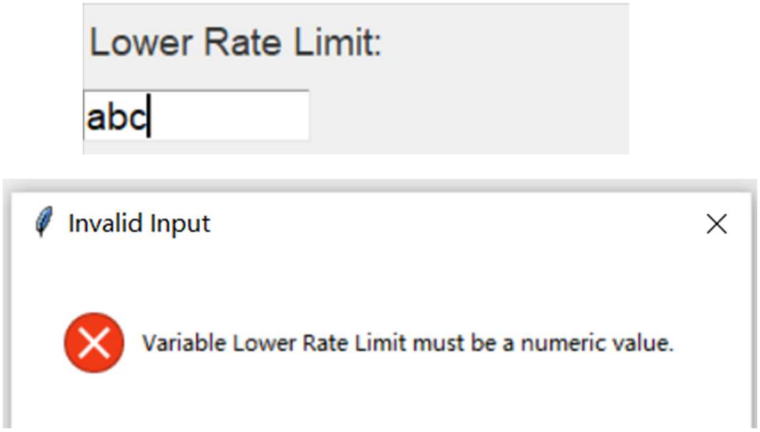
1. Start the application.
2. Register a new user (e.g., "new_user_5") and select the "AOO" mode.
3. Attempt to update variable values with invalid input, such as non-numeric characters or values outside valid ranges.
    - Set Lower Rate Limit to "abc."
    - Set Upper Rate Limit to "-10."
    - Set Atrial Amplitude to "2.5.1"
    - Set Atrial Pulse Width to "0.0"
    - Set ARP to "200.5"

Expected Results:

- An error message should appear for each invalid input.
- The user should not be able to update variables with invalid values.

Actual results:

- The error message occurred when the user input is not a numeric value.

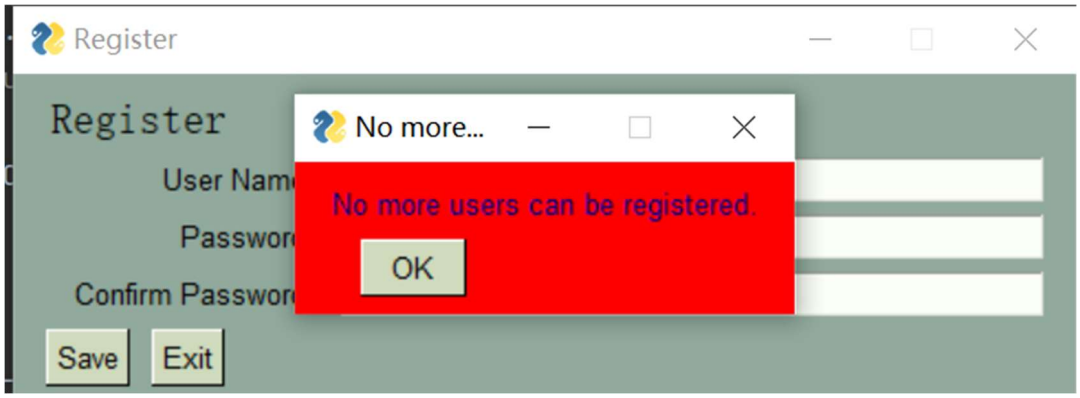


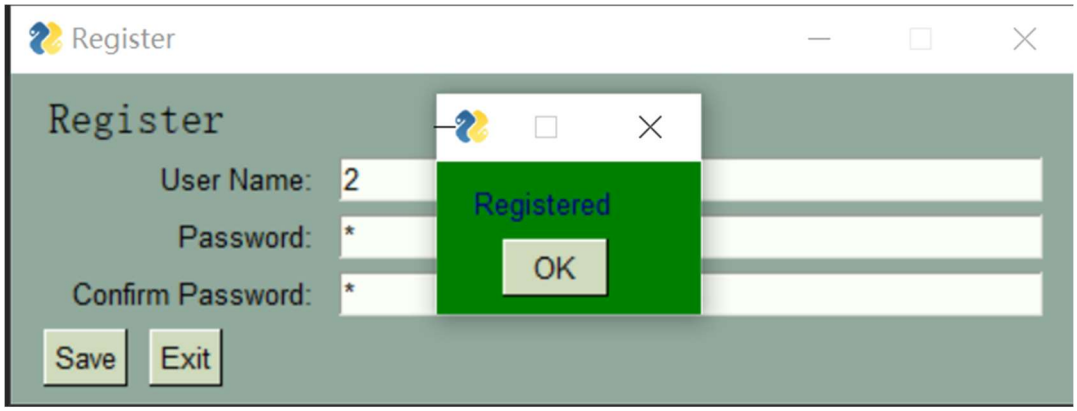
These detailed test cases cover various scenarios, including registration, login, mode selection, variable updates, mode-specific variables, and handling of invalid inputs.

Assignment 2 Test Cases:

1. Registering the max number of users

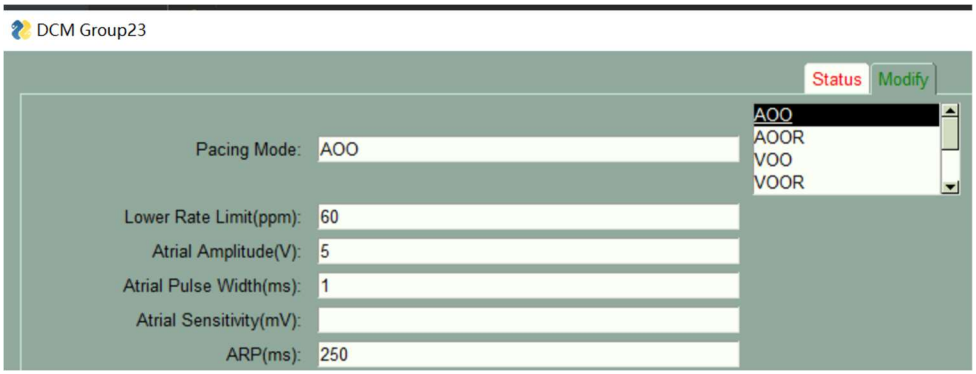
The purpose of this test is to check if the DCM prevents the user from registering more than 10 users. To begin, 10 users were created and stored in the users.json. Then, there was an attempt to register a new user (Figure 1.2). The expected output is that an error message appears alerting that the max number of users have already been registered and was matched by the actual output. The test result is a pass.

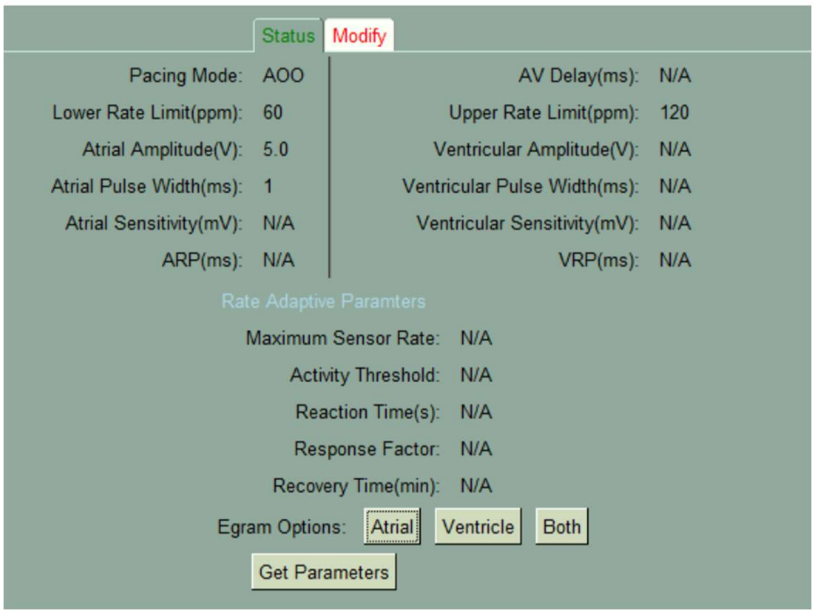




3. Active parameters

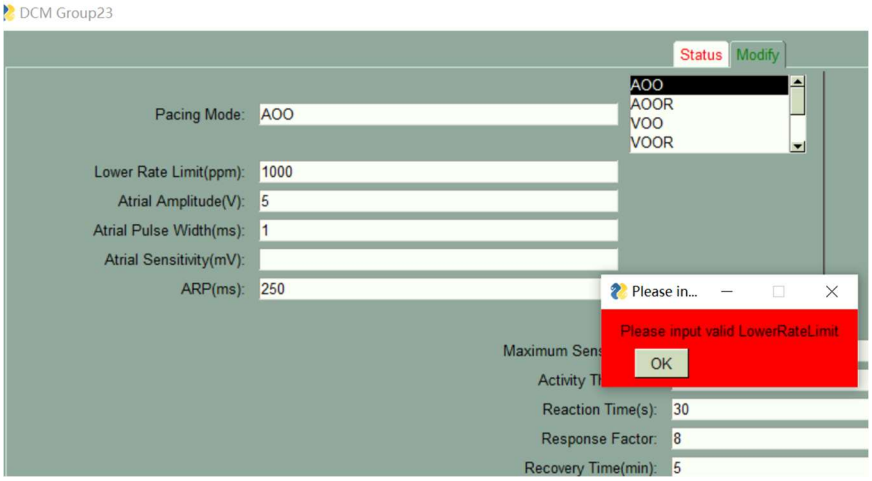
The active parameters should change according to the selected pacing mode. In this test, AOO was selected as the pacing mode and all required parameters were filled in the ‘Modify’ menu. The expected output is that in ‘Status’ menu, only the parameters as specified in are displayed. All other values were displayed with N/A. The actual output matched, and the test was repeated for all required pacing modes. The test result is a pass.

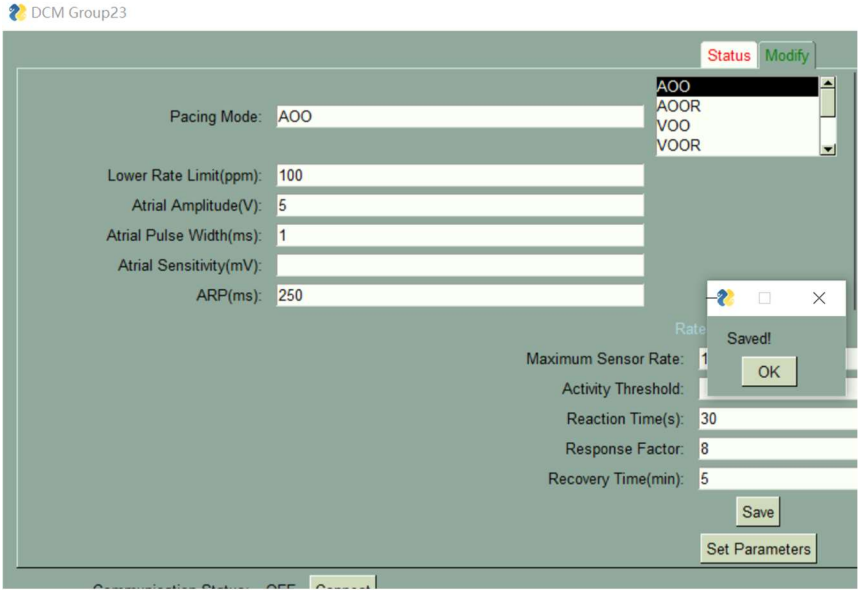




3. Range limitations

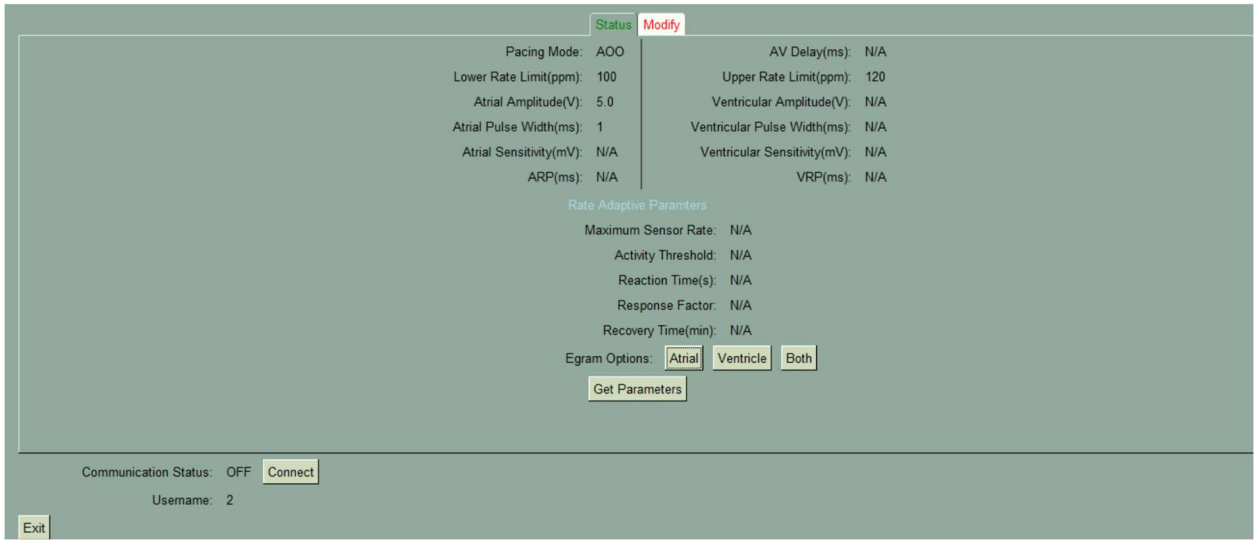
The purpose of this test is to ensure that the user inputted value is limited to a certain range for each parameter. The range limitations of the lower rate limit was tested using the exceedingly high value of 1000. Upon selecting ‘Save’, an error message should appear notifying that the input value is out of range and the change is not saved as expected. The actual result is as expected, and the test was repeated for all parameters using both high and low values. The test result is a pass.

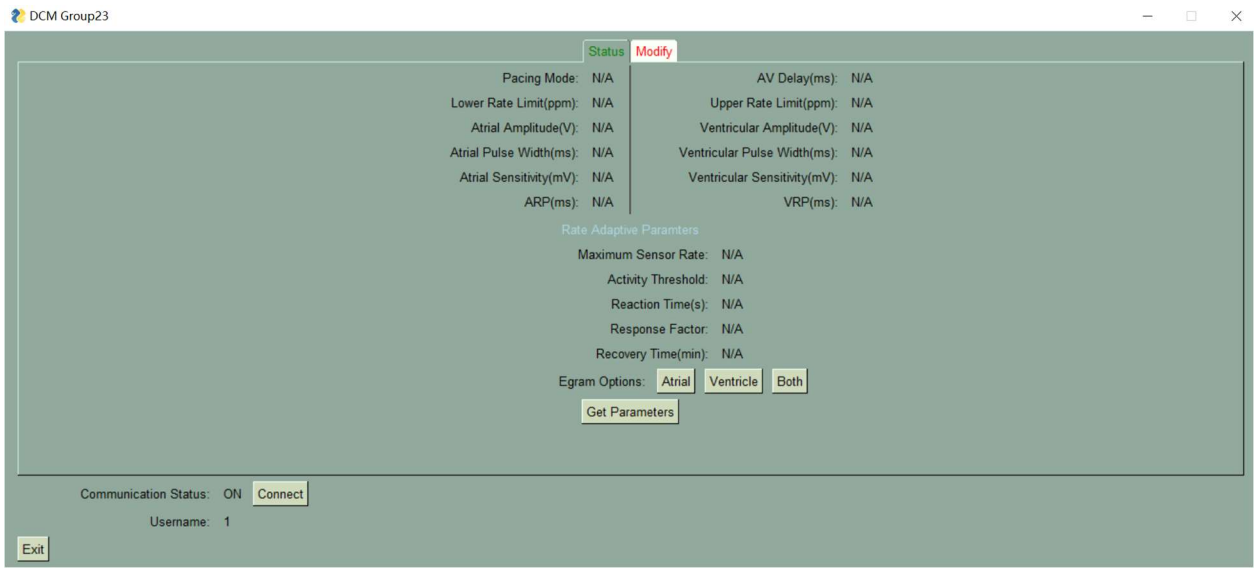




4. Connect with the pacemaker / Get the parameters in communication

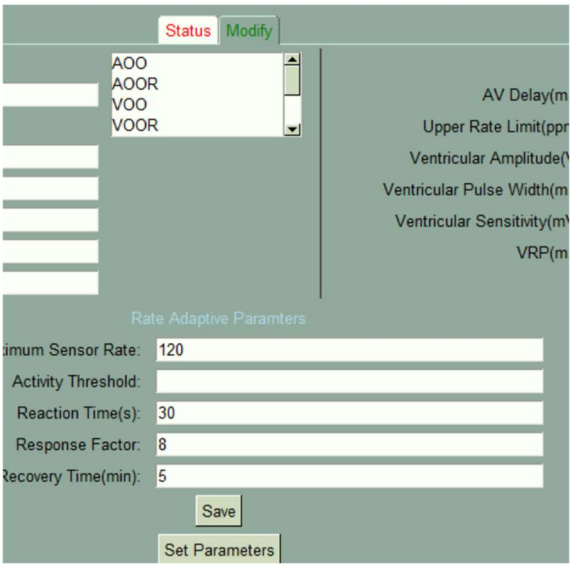
The purpose is to test whether DCM can get parameters from the pacemaker through UART. Input is the click on the connect button or get parameters button, both of which call the get_param_com function. The expected output is the current parameters showing on the status tab and the irrelevant parameters will be N/A. The actual output meets the expected one. The tests are passed for multiple trials.

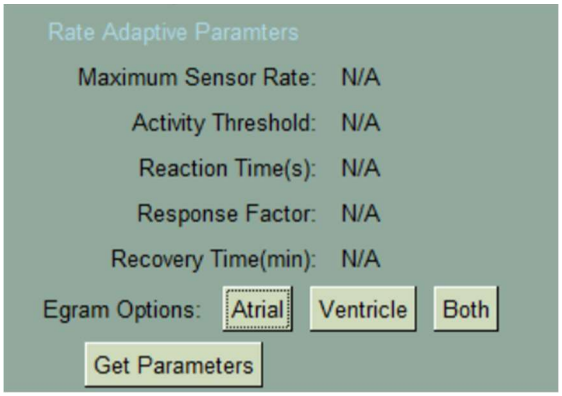




5. Set the parameters in communication

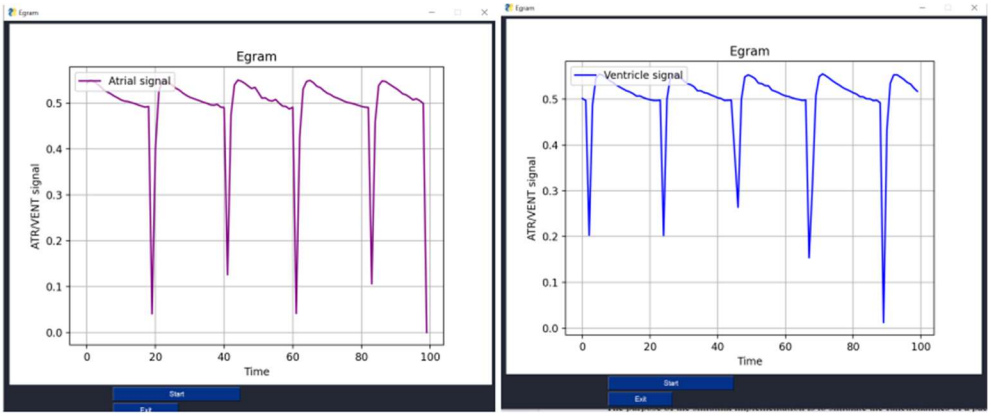
The purpose of this test is to make sure the parameters saved in DCM can be sent to the pacemaker successfully. The input is the click in the “Set Parameters” button in the modify tab. The expected output is a pop up window indicating that it was sent successfully. The actual output meets the expected one. All tests passed. It can be checked by clicking the “get parameters” button in the status tab after sending the parameters.

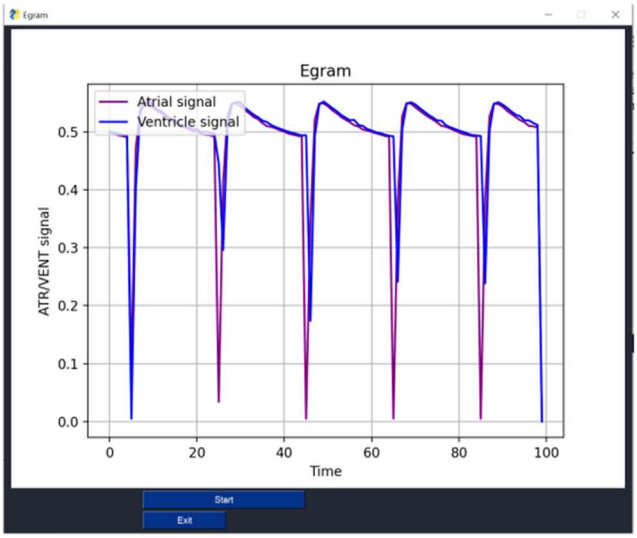




6. Egram display

This test is aimed to verify the egram of the pacemaker can be shown correctly. The input is clicking either atrial, ventricle or both, then clicking the start button in the egram window. The expected output is the animated graph in the egram window showing the signals from atrium and/or ventricles in real time. The actual output meets the expected one. The tests are all passed.





References

[1] McMaster University. (Fall 2023). PACEMAKER System Specification. [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/557175/viewContent/4323499/View?ou=557175

[2] McMaster University. (Fall 2023). What is a Pacemaker? [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/557175/viewContent/4324589/View

[3] McMaster University. (Fall 2023). Assignment2 [Online]. Available: https://avenue.cllmcmaster.ca/d2l/le/content/557175/viewContent/4380558/View