

# **SFWRENG 3K04 - L01 GROUP 40**

## **ASSIGNMENT 1**

**Hind Asaad**

**Michael Nasr**

**Qasim Sadaat**

**Eric Boardman**

**Michael Skinner**

**Venkat Kanagarajamuthaly**

# Table of Contents

<b>Table of Figures</b>	<b>4</b>
<b>Introduction</b>	<b>5</b>
<b>Simulink Requirements</b>	<b>5</b>
Current Requirements - Pacemaker	5
AOO Mode Requirements	8
VOO Mode Requirements	8
AAI Mode Requirements	9
VVI Mode Requirements	10
Requirement Changes	10
<b>DCM Requirements</b>	<b>12</b>
Welcome Screen	12
User Interface Capabilities	12
Telemetry Screen	12
Pacemaker Modes	12
Programmable Parameters	13
Real-Time Electrograms (egrams)	13
Communication with Pacemaker	13
Requirement Changes	14
Serial Communication:	14
DCM Utility Functions:	14
Printed Reports:	14
Strip Chart Recording Support:	14
<b>Simulink Model</b>	<b>15</b>
<b>DCM Module</b>	<b>32</b>
Introduction	32
Authentication Module	33
find_user(users, username)	33
login(users, username, password)	34
signup(users, username, password)	34
User Module	35
User Class	35
Class Constructor	35
Accounts Class	36
Class Constructor	37
load_accounts()	37
add_user(username, password)	38
update_device_file()	38
update_file()	39

Main Module	40
State Variables	40
clear_frame()	40
connection_UI()	40
show_welcome_state()	41
handle_login()	41
handle_signup()	42
show_telemetry_state()	42
update_text()	43
update_params(entries, key, errors, values, current_mode)	44
check_input(label_index, index, input_data, errors)	45
update_state(new_state)	45
<b>Simulink Design Process</b>	<b>46</b>
Decisions	46
Errors	48
Design Changes	50
<b>DCM Design Process</b>	<b>51</b>
Decisions	51
Errors	51
Design Changes	52
<b>Simulink Test Cases</b>	<b>52</b>
AOO Test Cases	52
VOO Test Cases	53
AAI Test Cases	55
VVI Test Cases	56
<b>DCM Test Cases</b>	<b>57</b>
Sign-in Page	57
Telemetry Page	60
<b>Conclusion</b>	<b>65</b>

# Table of Figures

Figure 1 - Simulink Model	15
Figure 2 - Programmable Simulink Parameters	16
Figure 3: Hardware Inputs and Pin Assignments	18
Figure 4: initial state parameters	19
Figure 5: Flow from initial state to mode 0 and mode 1	20
Figure 6: Flow from initial state to mode 3 and 4	20
Figure 7: Flow that ensures you can not enter an invalid mode.	20
Figure 8: Charging State for AOO	21
Figure 9: States' Edge Conditions and Pacing Conditions for AOO	22
Figure 10: Charging State for VOO	23
Figure 11: States' Edge Conditions and Pacing Conditions for the VOO	24
Figure 12: charging and sensing state AAI	25
Figure 13: State transition and recursion call back to charging and sensing state	26
Figure 14: AAI Sensing state	26
Figure 15: charging and sensing state VVI	27
Figure 16: State transition and recursion call back to charging and sensing state	27
Figure 17: VVI pacing state	28
Figure 18: Hardware Hiding of inputs	29
Figure 19: Hardware Hiding of outputs	30
Figure 20 - inputs of AAO State	47
Figure 21 - AAO State table when separate from other states	48
Figure 22 - AAO State table states	48
Figure 23 - List of all the old states added	49
Figure 24 - Figure of all the inputs where a few were not needed	49

# Introduction

For the decades since the invention of the pacemaker, there have been many discussions on the functionalities and needed improvements for the device. The array of problems a designer faces, from the multi-faceted role the heart plays in the body, the different situations and inputs of information from the body it reacts to, and the security issues with giving the ability to alter or stop a person's heartbeat with a device has made the topic an interesting problem for any designer, and that task is what document works to analyze and develop.

The development of a pacemaker device is an intertwining of hardware and software; writing commands to the NXP FRDM K-64, the pacemaker keeps a patient's heart beating in different modes. To develop a pacemaker that allows for these functionalities, an easy to use DCM is integral for any user. With that being said, the DCM and the Simulink stateflow work hand in hand for this working system, and this project was developed with two separate teams focusing on each problem.

Within the DCM, the basic aim was to develop a welcome screen allowing for a maximum of 10 different users to access information about the mode the pacemaker is currently in, while the Simulink portion develops 4 different modes for the pacemaker. This document discusses the behavior and functionality of each module undertaken, the design developed for said page, and the decision-making process for the design from the start to finish. Within the DCM, it covers every function and the decision making process behind on-screen functionalities, while the Simulink portion discusses intimately the process and understanding behind each mode and its implementation.

Overall, this document is an in-depth explanation of the functionalities of this pacemaker project, and gives a comprehensive understanding to any user.

## Simulink Requirements

### Current Requirements - Pacemaker

The Current requirements we were given are to use the pacemaker software to implement modes AOO, VOO, AAI and VVI based on the conditions of the heart. The conditions of the heart being the input data. See Table below.

Parameters	Units	Accepted Values	Description
Lower Rate Limit (LRL)	ppm	30 - 175	This parameter assumes a critical role in pacemaker programming, signifying the threshold for the minimum heart rate at which the pacemaker intervenes to

			<p>enforce rhythm stability. In non-adaptive modes, it serves as the steadfast baseline heart rate, ensuring cardiac regularity. This feature is particularly pivotal in cases of bradycardia or similar arrhythmias, as it guarantees a constant, medically designated heart rate. In essence, the Lower Rate Limit acts as a safety mechanism, assuring that the heart maintains a consistent rhythm when the intrinsic electrical system experiences disruptions, thereby upholding the patient's cardiovascular well-being.</p>
Upper Rate Limit (URL)	ppm	30 - 175	<p>The "Upper Rate Limit" parameter is a pivotal component of pacemaker settings, designating the maximum allowable heart rate at which the pacemaker will intervene. This parameter is essential in cases of tachyarrhythmias or other conditions where an excessively rapid heart rate must be controlled. It operates as a safeguard against excessively high heart rates, ensuring that the pacemaker effectively regulates and caps the upper limit of the heart's rhythm. In summary, the Upper Rate Limit serves as an important control mechanism to prevent the heart from exceeding a predefined threshold, promoting cardiac stability and safety, particularly in situations where the heart's intrinsic electrical system exhibits excessive acceleration.</p>
Mode	N/A	AOO, VOO, AAI, VVI	<p>Indicates which operating mode the pacemaker should operate in</p>
Pulse Amplitude	V	0.5 - 5V	<p>The magnitude or strength of the electrical pulse transmitted to either the atrium or ventricle during heart pacing.</p>
Pulse Width	ms	0.05 - 1.9	<p>The pulse width, or duration, of the electrical signals delivered to either the</p>

			atrium or ventricle when pacing the heart.
vent_cmp_detect	U_int	0 or 1	<p>The binary notation of "0" and "1" utilized in this context represents a fundamental aspect of pacemaker operation, particularly in scenarios where the sensed ventricular pulse aligns with the Lower Rate Limit or falls below it. When the pacemaker registers a "0," it signifies that no electrical signal from the ventricle has been detected at the Lower Rate Limit. This absence of ventricular activity prompts the pacemaker to initiate a pacing stimulus to ensure that the heart maintains the prescribed minimum heart rate, a critical function in preventing bradycardia or other cardiac arrhythmias. Conversely, when the pacemaker records a "1," it indicates that the sensed ventricular pulse is indeed present at or below the Lower Rate Limit. In such cases, the pacemaker refrains from delivering additional pacing stimuli, allowing the patient's intrinsic cardiac activity to prevail, thus ensuring that artificial pacing is only administered when necessary to maintain appropriate heart rates and cardiac stability. This binary distinction is central to the pacemaker's responsive and adaptive functioning, optimizing patient care based on real-time heart activity.</p>
atr_cmp_detect	U_int	0 or 1	<p>The binary representation of "0" and "1" is a fundamental element in pacemaker functionality, specifically when assessing the atrial activity in relation to the Lower Rate Limit. When the pacemaker registers a "0," it indicates the absence of an atrial signal at or below the Lower Rate Limit. In response to this absence, the pacemaker generates a pacing impulse to ensure that the heart maintains the prescribed minimum atrial rate, a crucial measure to prevent</p>

			bradycardia and other cardiac rhythm irregularities. Conversely, when the pacemaker records a "1," it signifies that the atrial signal is detected at or below the Lower Rate Limit. In such instances, the pacemaker refrains from delivering additional pacing stimuli, enabling the patient's intrinsic atrial activity to dictate the heart's rhythm. This binary distinction plays a pivotal role in the pacemaker's adaptability, guaranteeing that artificial pacing is initiated only when essential to uphold a suitable heart rate and cardiac stability based on real-time atrial dynamics.
--	--	--	--

## AOO Mode Requirements

The AOO mode, a vital component of pacemaker configuration, is characterized by the consistent pacing of the atrium at predefined intervals, irrespective of the heart's intrinsic electrical activity. In this mode, the Lower Rate Limit holds a central role as it sets the threshold for the atrial pacing rate. Precise control over pulse parameters is maintained in this mode, where two key factors come into play: Pulse Width and Amplitude. Pulse Width governs the temporal duration of the electrical impulses delivered to the atrium, ensuring accuracy in pacing timing, while Amplitude dictates the strength and magnitude of these electrical pulses. The AOO mode functions autonomously, without regard to the heart's prevailing conditions, thereby ensuring a reliable pacing strategy that adheres to the prescribed Lower Rate Limit. This consistent atrial pacing mechanism is vital for addressing cardiac issues effectively, as it sustains a stable and controlled heart rate.

Inputs	Actions
The equation $LRL - pw$ determines period between atrial pulses	<ul style="list-style-type: none"> <li>- Enter the Pacing state</li> <li>- Pace state Pulse the Atrium</li> <li>- Return to the Charging State</li> </ul>

## VOO Mode Requirements

The VOO mode, a fundamental aspect of pacemaker functionality, is characterized by consistent ventricular pacing at predetermined intervals, irrespective of the heart's intrinsic



electrical activity. In this mode, the Lower Rate Limit is the pivotal parameter that defines the pace at which the ventricle is stimulated. The pulse attributes within VOO mode are meticulously governed by two critical factors: Pulse Width and Amplitude. Pulse Width dictates the temporal duration of the electrical impulses delivered to the ventricle, ensuring precise pacing timing, while Amplitude defines the strength and magnitude of these electrical pulses. Collectively, these parameters enable the VOO mode to sustain a steady ventricular rhythm, with the Lower Rate Limit serving as the cornerstone for ventricular pacing, which ensures that the heart's rate remains constant and conforms to medically prescribed criteria. This mode operates independently of the heart's intrinsic conditions, providing consistent pacing to address various cardiac issues efficiently.

Inputs	Actions
The equation $LRL - pw$ determines period between ventricle pulses	<ul style="list-style-type: none"> <li>- Enter the Pacing state</li> <li>- Pace state Pulse the Ventricle</li> <li>- Return to the Charging State</li> </ul>

## AAI Mode Requirements

The AAI pacing mode, an intrinsic component of pacemaker programming, primarily involves atrial pacing to synchronize with or elevate the atrial rate if it falls below the prescribed Lower Rate Limit. The intricacies of this mode's operation are determined by the precise calibration of two key parameters: Pulse Width and Amplitude. Pulse Width dictates the duration of the electrical impulse delivered to the atrium, optimizing the pacing's temporal aspects, while Amplitude governs the magnitude or strength of the electrical pulses. The combination of these settings enables the AAI mode to maintain cardiac rhythm within the desired range, thereby ensuring that the patient's heart rate adheres to the Lower Rate Limit, and effectively preventing bradycardia and its associated complications through precise and controlled electrical stimulation of the atria.

Inputs	Actions
The equation $LRL - pw$ determines period between ventricle pulses	<ul style="list-style-type: none"> <li>- Enter the Pacing state</li> <li>- Pace state Pulse the Ventricle</li> <li>- Return to the Charging State</li> </ul>
If $atr\_cmp\_detect == 0$ And after $LRL - pw$	<ul style="list-style-type: none"> <li>- Enter the Pacing State</li> <li>- Pulse the Ventricle</li> <li>- Return to the Charging State</li> </ul>

If atr_cmp_detect == 1	<ul style="list-style-type: none"> <li>- Ventricle pulse has been sensed</li> <li>- Return to Charging State</li> </ul>
------------------------	---

## VVI Mode Requirements

The VVI mode, a fundamental component of pacemaker configuration, operates by pacing the ventricle with a view to matching the Lower Rate Limit if the heart's intrinsic ventricular electrical activity falls below this specified threshold. It essentially functions as a safety net to ensure that the ventricular rhythm adheres to or surpasses the preset Lower Rate Limit. The intricate aspects of pacing within this mode are contingent on two critical parameters: Pulse Width and Amplitude. Pulse Width dictates the temporal duration of the electrical impulses transmitted to the ventricle, ensuring precise control over pacing timing. On the other hand, Amplitude governs the strength and intensity of these electrical pulses. The combination of these parameters empowers the VVI mode to sustain a stable ventricular rhythm, essentially acting as a guardian against excessively low ventricular rates. This is particularly crucial in cases where the heart's intrinsic electrical system exhibits insufficiencies, providing controlled and consistent electrical stimulation to maintain the desired ventricular rate.

Inputs	Actions
The equation $LRL - pw$ determines period between ventricle pulses	<ul style="list-style-type: none"> <li>- Enter the Pacing state</li> <li>- Pulse the Ventricle</li> <li>- Return to the Charging State</li> </ul>
If vent_cmp_detect == 0 And after $LRL - pw$	<ul style="list-style-type: none"> <li>- Enter the Pacing State</li> <li>- Pulse the Ventricle</li> <li>- Return to the Charging State</li> </ul>
If vent_cmp_detect == 1	<ul style="list-style-type: none"> <li>- Ventricle pulse has been sensed</li> <li>- Return to Charging State</li> </ul>

## Requirement Changes

As a team, we are anticipating that by the end of the project there will be a major change in the amount of modes we are expected to handle, the parameters we will be working with will increase as well, and we will be using serial communication to interact with the DCM application.

### Additional Modes:

The modes will be expected to handle the following: DDDR, VDDR, DDIR, DOOR, VOOR, AOOR, VVIR, AAIR, DDD, VDD, DDI, DOO, VVT, and AAT. Transitioning from a system with

four basic pacing modes, including AOO, VOO, VVI, and AAI, to a more complex set involving DDDR, VDDR, DDIR, DOOR, VOOR, AOOR, VVIR, AAIR, DDD, VDD, DDI, DOO, VVT, and AAT requires meticulous planning and careful implementation. The expanded array of pacing modes brings versatility and precision in addressing various cardiac conditions. To manage this transition effectively, it's essential to have a deep medical understanding of each mode and their uses, their indications, and the relevant parameter settings. Additionally, updating and validating the device programming in line with individual patient needs is crucial. Routine monitoring, documentation, and post-transition evaluation are also vital to ensure that the selected modes provide optimal cardiac care while minimizing complications. A well-structured and gradual transition strategy, along with constant communication ensures a seamless shift to these advanced pacing modes while ensuring that the new requirements do not hinder the performance of the pacemaker system.

The transition to the more complex set of pacing modes not only broadens the range of available modes but also necessitates the implementation of a dual sensing mode. In a dual sensing mode, the pacemaker system can simultaneously monitor and respond to electrical signals from both the atrium and the ventricle, providing a more comprehensive and nuanced assessment of the patient's cardiac activity. This dual sensing capability allows the device to tailor its responses to the specific needs of the patient, adapting to changes in atrial and ventricular activity as required by the selected pacing mode. The incorporation of this dual sensing feature is a crucial enhancement, as it enables the pacemaker system to function with greater precision and responsiveness, further optimizing patient care.

### Serial Communication:

Implementing serial communication within a system that offers an extensive range of pacing modes and dual sensing capabilities can be challenging due to several factors. Firstly, the increased complexity of the system requires a sophisticated and robust communication protocol to handle the transmission and reception of data accurately. The higher number of modes and sensing channels means that there's more data to manage and synchronize, which can lead to potential complications in data integrity. Additionally, ensuring that serial communication remains reliable and real-time, especially in a medical context, is crucial. The system needs to maintain precise timing and data synchronization to ensure that the pacemaker responds appropriately to a potential patient's cardiac needs. As any delays or errors in serial communication can have significant consequences for a potential patient's health. Furthermore, the system must be designed to handle potential interferences and electromagnetic compatibility issues, which can be particularly challenging in a medical environment where various electronic devices are in use. Lastly, maintaining data security and patient privacy is of utmost importance. The implementation of robust encryption and authentication mechanisms is necessary to protect sensitive medical data from unauthorized access.

### Parameters:

Effectively managing a significant number of input parameters in the firmware of a pacemaker involves systematic categorization and prioritization of parameters based on their clinical significance. Establish default values for parameters and allow for parameter profiles to be saved

and loaded. Provide comprehensive documentation to explain the purpose, acceptable values, and potential consequences of parameter changes. Implement rigorous testing, gather feedback, and continuously improve the firmware's usability. Automate parameter adjustments when applicable to reduce manual input, and introduce search and filter functions for easy access to specific parameters. Lastly, our documentation will have to explain to users how to become proficient in navigating and configuring the firmware's numerous parameters, ultimately enhancing the pacemaker's performance and ease of use.

## DCM Requirements

### Welcome Screen

The welcome screen is an interface that must allow new users to register and returning users to log in. The total number of registered users can not go over 10.

### User Interface Capabilities

1. The windows must properly display the text and graphics for the user interface regardless of the window size.
2. User text input and button events must be handled
3. Windows must seamlessly transition to one another with no remnants of the previous window showing on the new window
4. The user must be able to see all relevant information, and the information must be correct

### Telemetry Screen

The telemetry screen shows the current mode that the user is in, a dropdown menu that allows the user to select the pacemaker mode, the corresponding parameters for that mode and their values, and text boxes so the user can change the parameters. There is also a button to submit changes, so the user can save the changes they've made.

### Pacemaker Modes

There are four modes currently implemented: AOO, VOO, AAI, and VVI. These modes are accessed through the previously mentioned dropdown menu on the telemetry screen.

## Programmable Parameters

There are 8 programmable parameters that we have implemented so far, the information regarding each one is documented below:

Parameter	Unit	Nominal	Valid Ranges	Increments	Modes
Lower Rate Limit	ppm	60	30-50 50-90 90-175	5 1 5	AOO, VOO, AAI, VVI
Upper Rate Limit	ppm	120	50-175	5	AOO, VOO, AAI, VVI
Atrial Amplitude	V	3.5	0.5-3.2 3.5-7.0	0.1 0.5	AOO, AAI
Atrial Pulse Width	ms	0.4	0.05 0.1-1.9	- 0.1	AOO, AAI
Ventricular Amplitude	V	3.5	0.5-3.2 3.5-7.0	0.1 0.5	VOO, VVI
Ventricular Pulse Width	ms	0.4	0.05 0.1-1.9	- 0.1	VOO, VVI
VRP	ms	320	150-500	10	VVI
ARP	ms	250	150-500	10	AAI

## Real-Time Electrograms (egrams)

At this point in our project, we have not implemented the electrogram functionality as there is currently no communication with the pacemaker and DCM. However, we have developed the data type. Our current plan, which could change, is to sample from the pacemaker after a certain timestep. Every timestep, we measure all of the egram data and store that as an array/vector. These individual arrays will then be stored in a larger array. In short, we will be using a two-dimensional array, where each internal array measures the values at a specific time.

## Communication with Pacemaker

Although the DCM is not communicating with the pacemaker yet, we've made provisions for some functionalities that will be fully implemented once we start the communication. In the DCM

GUI, the user can see a label at the top saying whether or not the DCM is detecting a device, it also displays whether the connected device is the same device that was last connected.

## Requirement Changes

By the end of the project, the DCM will include full communication with the pacemaker, all egrams data available to the user, all user reports, and more.

### Serial Communication:

The DCM will be able to send and receive information from the pacemaker by using the functionalities developed so far with the programmable parameters. As previously mentioned, all the DCM can do right now is set and save the programmable parameters. By the time the project is fully completed, these programmable parameters will be able to control what is happening on the pacemaker. Conversely, the pacemaker will also be able to send data to the DCM to be viewed by the user. This will be done with the electrogram data, and it will be stored through the methods outlined previously.

### DCM Utility Functions:

An About function will be created that displays the application, model number, application software revision number, DCM serial number, and institution name. On top of this, the user will be able to set the date and time of the device through a function on the DCM GUI. Furthermore, the New Patient function will be able to interrogate a new device without exiting the software. To expand on what was mentioned previously, we will use the method that detects if a new device is connected to do this. If a new device is connected, we need to make sure to recognize that and keep the program going. Finally, a Quit function will be implemented to end the telemetry session, aside from just clicking the “x” button in the corner of the GUI

### Printed Reports:

We will also add the ability to display several parameter and status reports on screen at the user’s request, including a Bradycardia Parameters Report, a Temporary Parameters Report, and several more. On top of this, we will add the ability to display a Rate Histogram Report and a Trending Report so the user can view what is happening visually instead of just through written reports.

### Strip Chart Recording Support:

The DCM will be able to display real-time ECG data once it’s connected to the pacemaker. The DCM will be able to display up to three real-time traces in an easy-to-interpret manner.

# Simulink Model

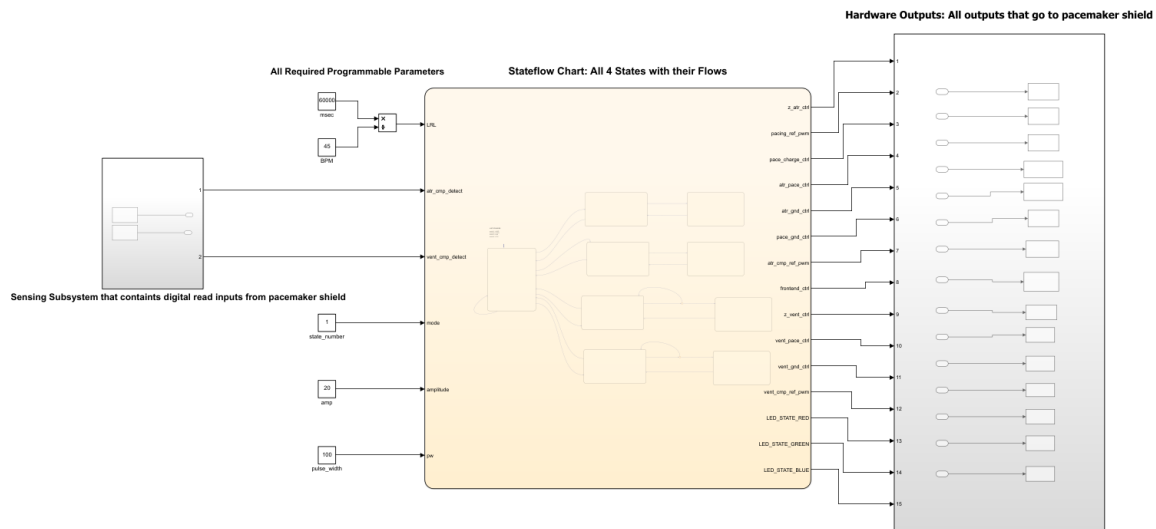


Figure 1 - Simulink Model

The overview of our Simulink model shown above describes the implementation of a 4-mode Pacemaker system, wherein the choice of mode is determined by input parameters from the requirements section, governing the electrical pulse characteristics delivered to the heart. Once a mode is selected, the pacemaker shield is configured with the mode's parameters, maintaining that mode until a new one is programmed. These modes are represented by states, continuously outputting data corresponding to the active state. The outputs are then channeled through the pacemaker shield, where they undergo specific circuitry discussed later in this section. Signals from this circuitry are subsequently routed to HeartView, where the corresponding electrocardiogram for the set mode is displayed. The HeartView software facilitates testing of the pacemaker shield with diverse heart conditions, sometimes employing these conditions as inputs on our board.

## Programmable Inputs

## All Required Programmable Parameters

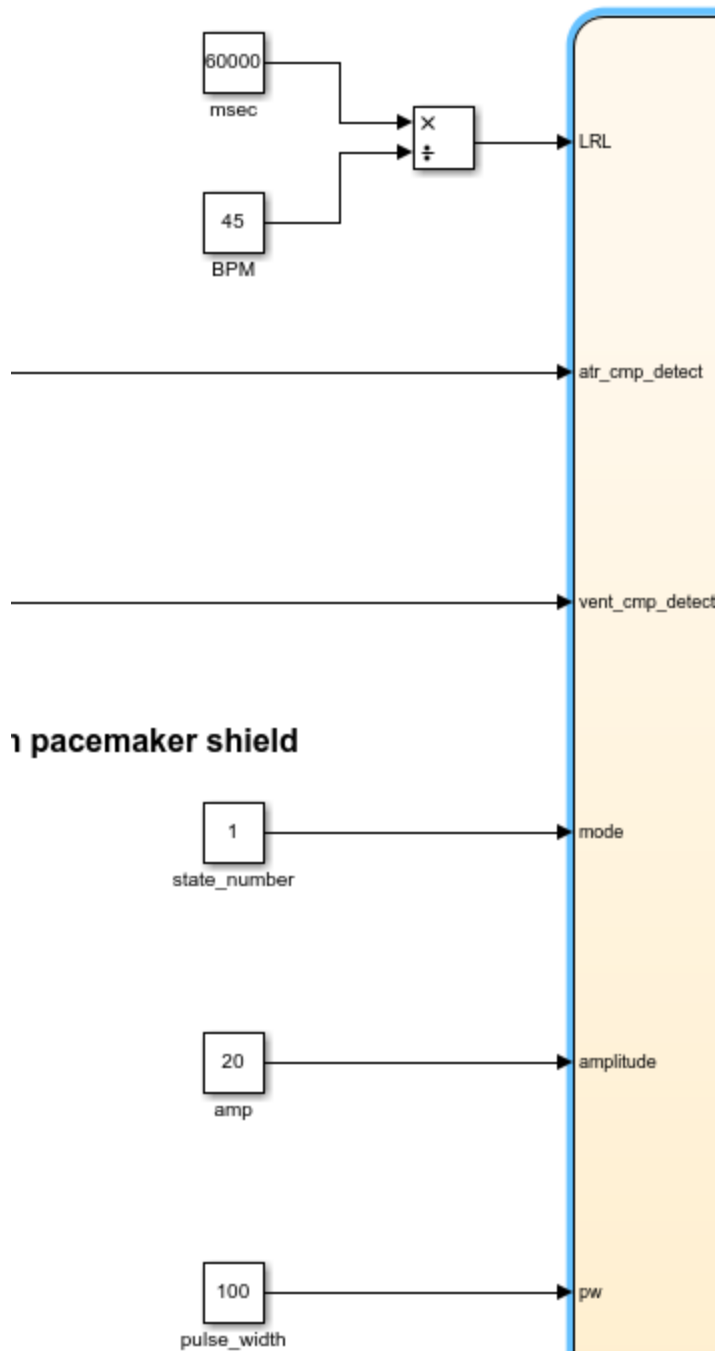


Figure 2: Programmable Simulink Parameters

Here is an overview of each input and its functionality within the system:

Parameter Name	Functionality	Initialization Value
----------------	---------------	----------------------



LRL (Lower Rate Limit)	<p>This is a measure of the synthetic pulses per minute. The LRL specifically is the bpm in which if the heart pulse is lower than this value, the pacemaker will send a pulse depending on the sensing and response to sensing values. The lower rate limit is achieved through the follow expression:</p> <p>LRL = nominal bpm / 1 minute.</p>	<p>45 ppm</p> <p>The nominal value is 60 ppm, it is currently set to 45 for testing purposes.</p>
mode	<p>Modes are set as follows:  1 = AOO  2 = VOO  3 = AAI  4 = VVI</p> <p>This variable tracks the mode that the user sets the pacemaker to behave according to.</p>	1
amplitude	<p>This is the amplitude of the synthetic pulse that is sent to the heart.</p> <p>The amplitude is achieved through the expression:</p> <p>Amplitude = (actual voltage / max voltage) * 100%</p>	<p>20 mV</p> <p>Actual Voltage is a range from 0-3.3 V</p> <p>Max Voltage is 5 V</p>
pw (Pulse Width)	Length of the Pulse	100

## Hardware Inputs

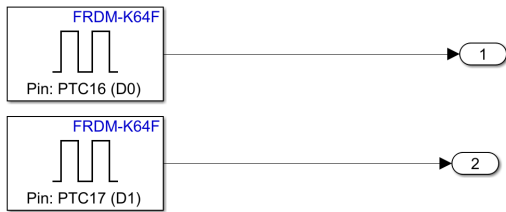


Figure 3: Hardware Inputs and Pin Assignments

Parameter Name	Pin Assignment	Value	Function
atr_cmp_detect	PTC16 (D0)	1 bit signal (0 or 1)	If the signal is above a certain threshold a 1 bit will be driven on the line meaning that a pulse in the atrium has occurred Else the line will drive 0.
vent_cmp_detect	PTC17 (D1)	1 bit signal (0 or 1)	The same functionality of the atr_cmp_detect but for the ventricle

### Initial State

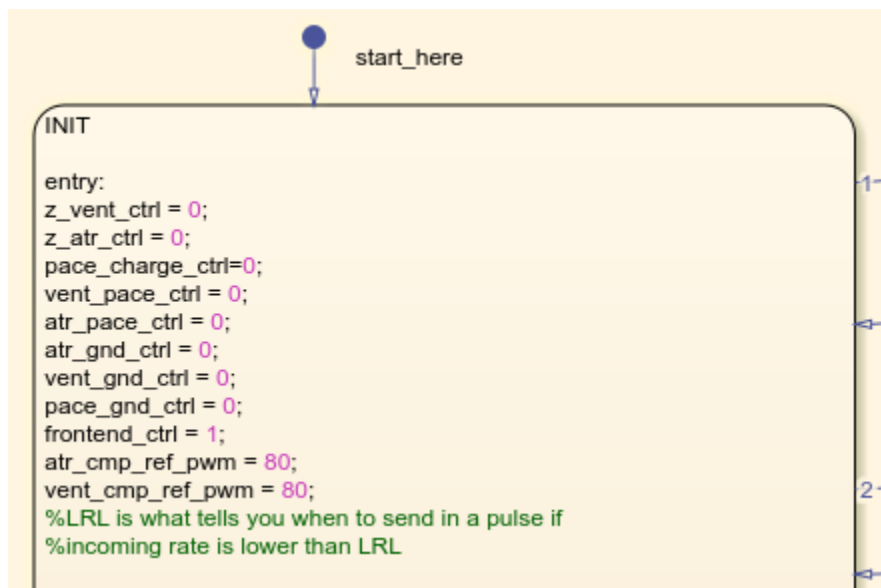


Figure 4: initial state parameters

This initial state sets it so that the impedance circuit is attached and ensures that the board will not be short circuited or fried. It is a state that does not do anything, no charging, or discharging.

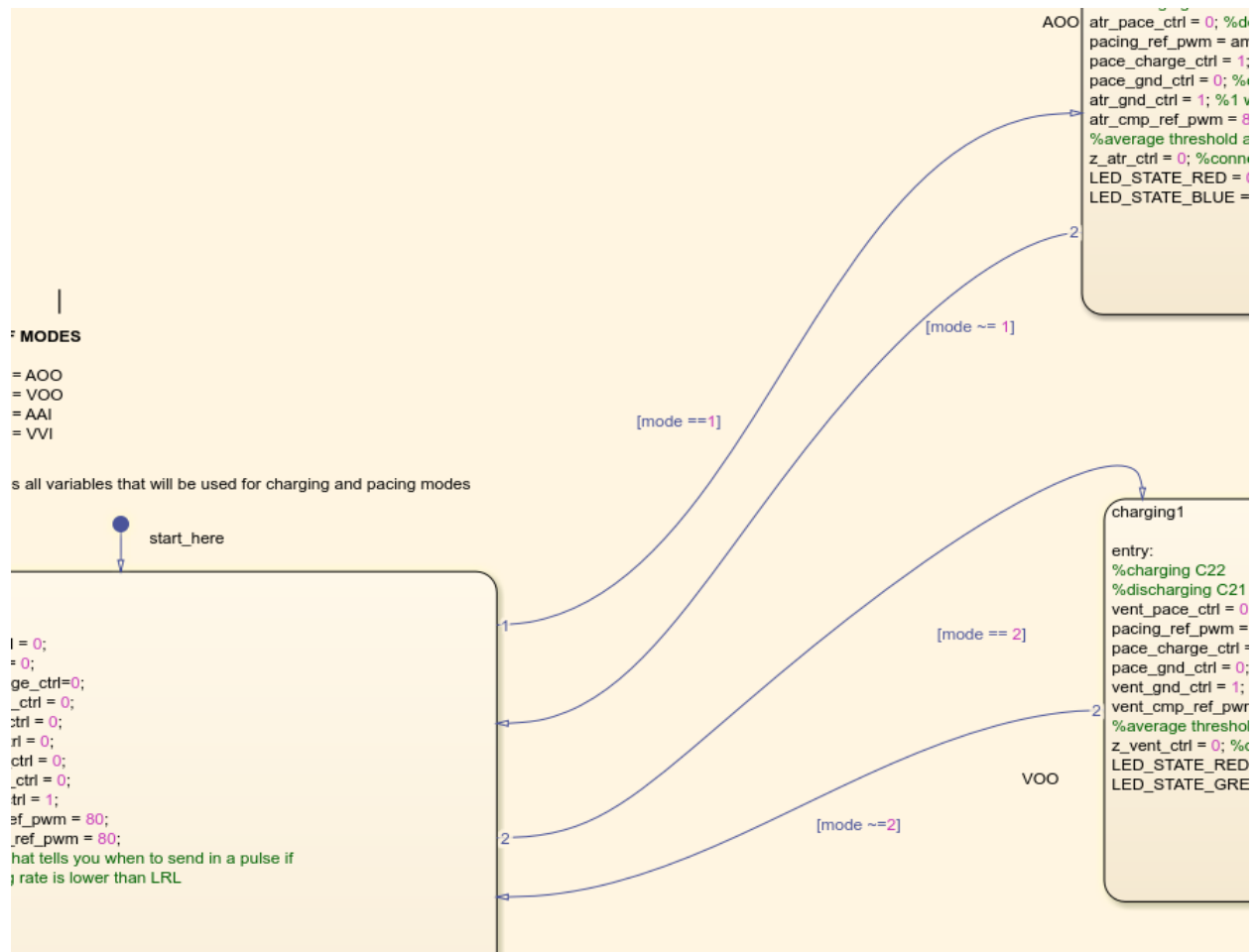


Figure 5: Flow from initial state to mode 0 and mode 1

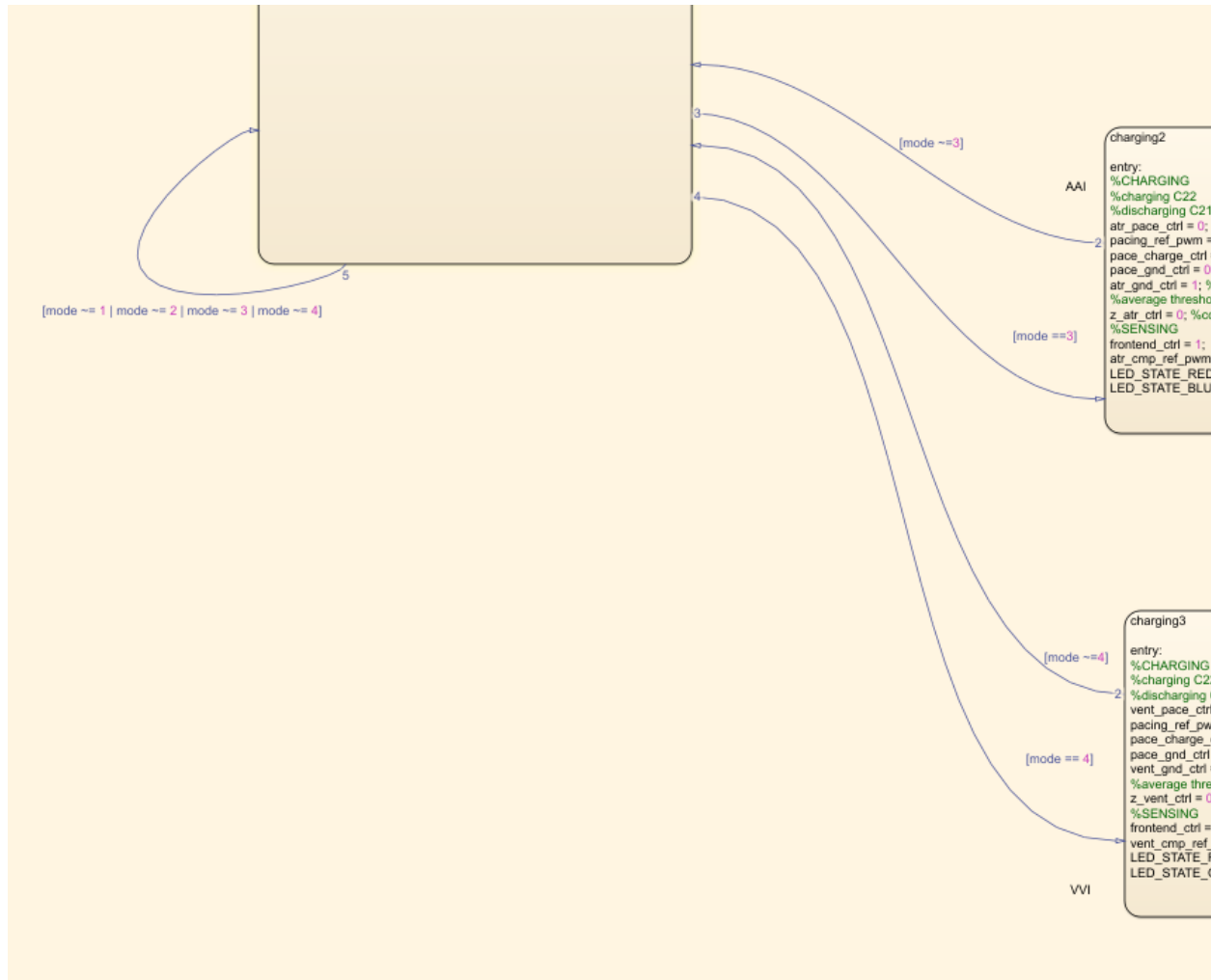


Figure 6: Flow from initial state to mode 3 and 4.

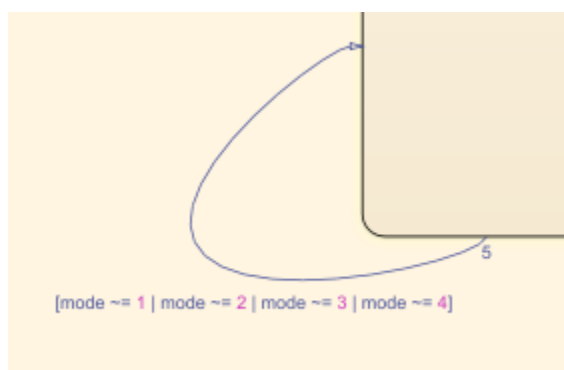


Figure 7: Flow that ensures you can not enter an invalid mode.

These flows from the initial state to the preceding states are implemented in a way such that the program has somewhere to follow if it is not the current mode that it is at. For example, if the user inputs mode 3, simulink traverses both modes 1 and 2 before it gets to mode 3. By including the statements “mode ~= x”, this gives the model a way to backtrack to the initial state

if it is not mode x. In the case of inputting mode 3, the model goes to mode 1 then back to the initial, then mode 2 and back to the initial, then it settles on mode 3.

The recursive statements [mode ~=1 | mode ~=2 | mode ~=3 | mode ~=4] ensures that the user cannot implement an invalid mode.

## States AOO & VOO

### AOO

**Brief Description:** This state does not listen for any sensing feedback for the heart as it has no response to sensing information. It will send a synthetic pulse regardless of the feedback.

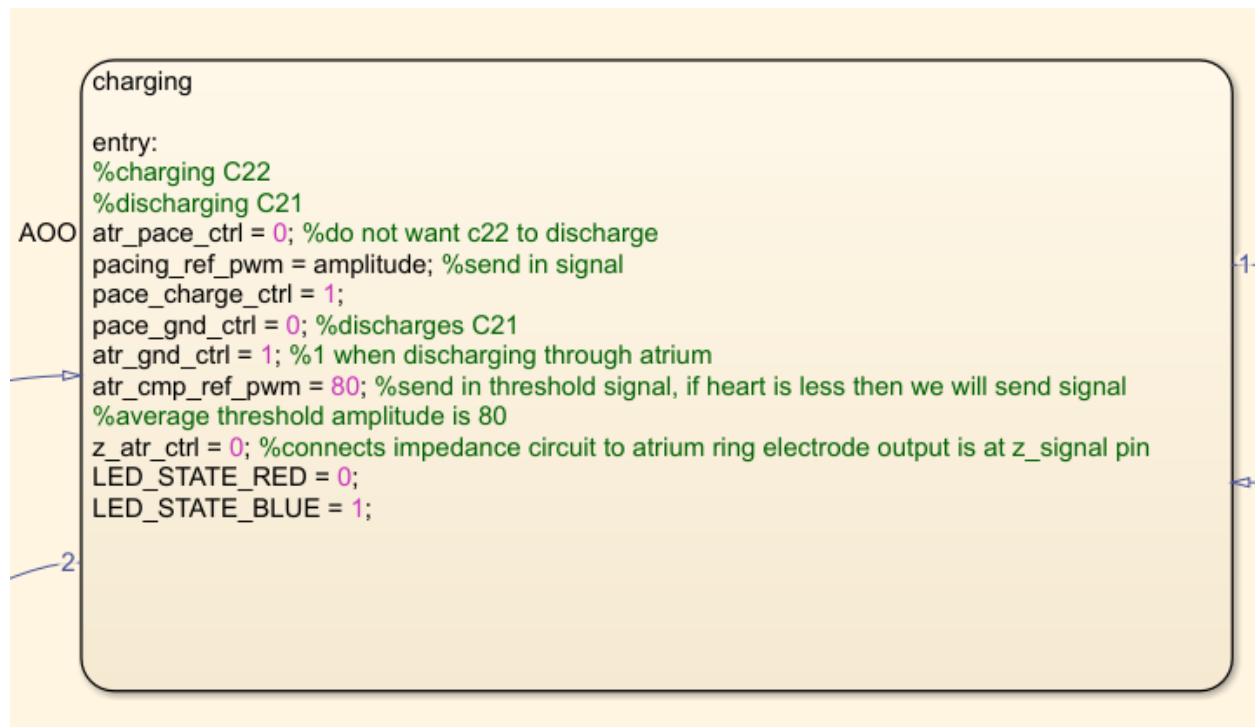


Figure 8: Charging State for AOO

Once arriving at this state a few main changes occur and that would be setting atr\_pace\_ctl to 0, atr\_gnd\_ctl to 1, pac\_gnd\_ctl to 1 and pace\_charge\_ctl to 1 as we want to make sure that capacitor C22 is charging and C21 is discharging. We also have the red led OFF and the blue led ON.

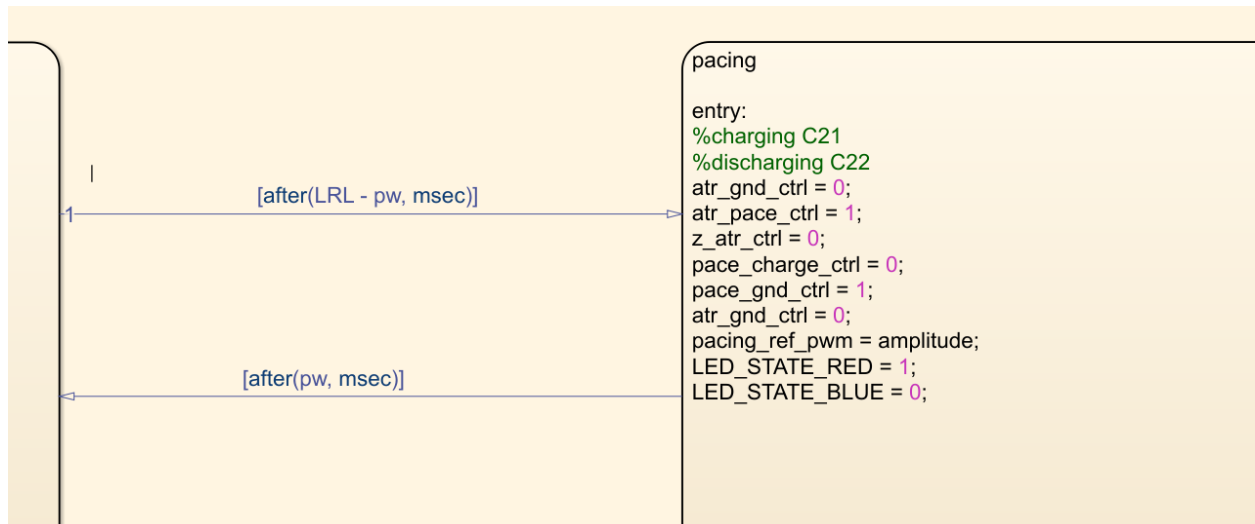


Figure 9: States' Edge Conditions and Pacing Conditions for AOO

The edge from charging to pacing state waits until the Lower Rate Limit - PW milliseconds as that will be that time until a pulse should be sent out as on cycle will be LRL + PW milliseconds. We then set the variable so that the Capacitor C21 is now the charging capacitor and Capacitor C22 is discharging which sends the signal that paces the heart. Red leds are set to ON and the Blue leds are set to OFF. At this state the signal would be sent once every cycle.

## VOO

**Brief Description:** This state does not listen for any sensing feedback for the heart as it has no response to sensing information. It will send a synthetic pulse regardless of the feedback.

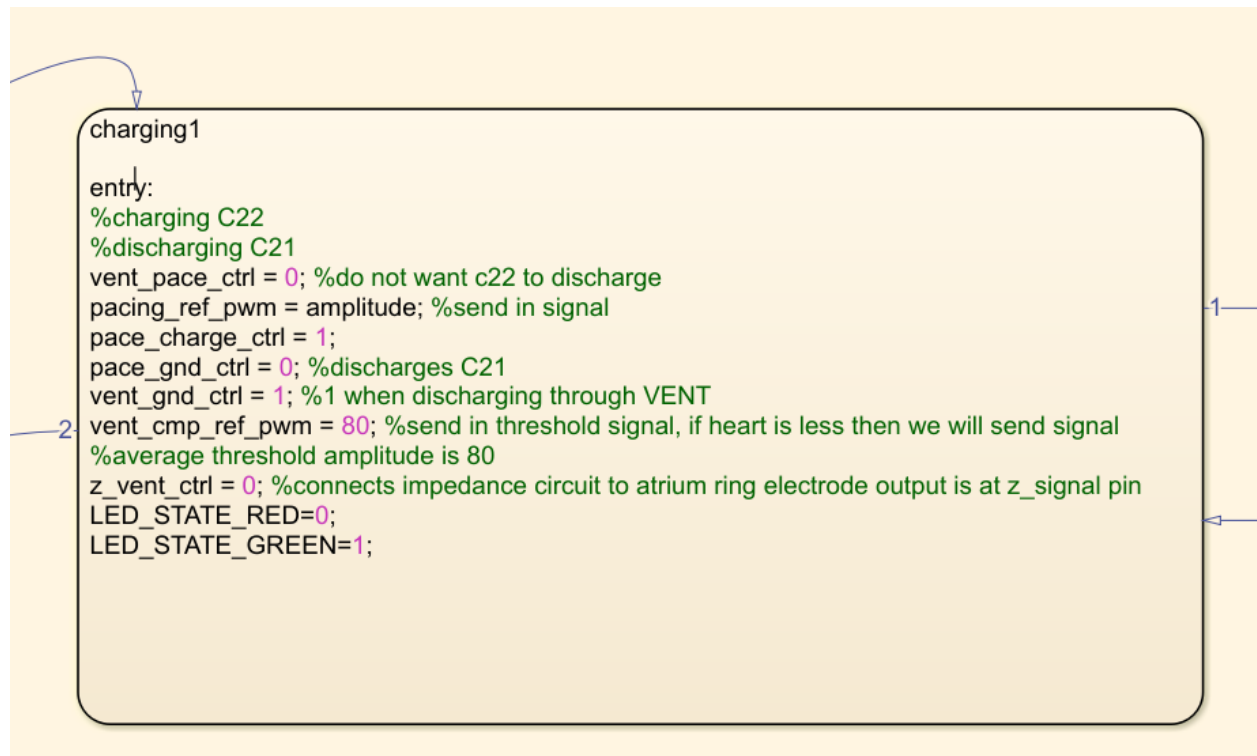


Figure 10: Charging State for VOO

Similar to AOO, Once it arrives at this state, vent\_pace\_ctrl will be set to 0, vent\_gnd\_ctrl to 0, pac\_gnd\_ctrl to 1 and pace\_charge\_ctrl to 1. This is to make sure that we went to charge capacitor C22 to get ready to send as we want to make sure that capacitor C22 is charging and 1 is discharging. The VOO charging state sets the red led to off and the green led to on.

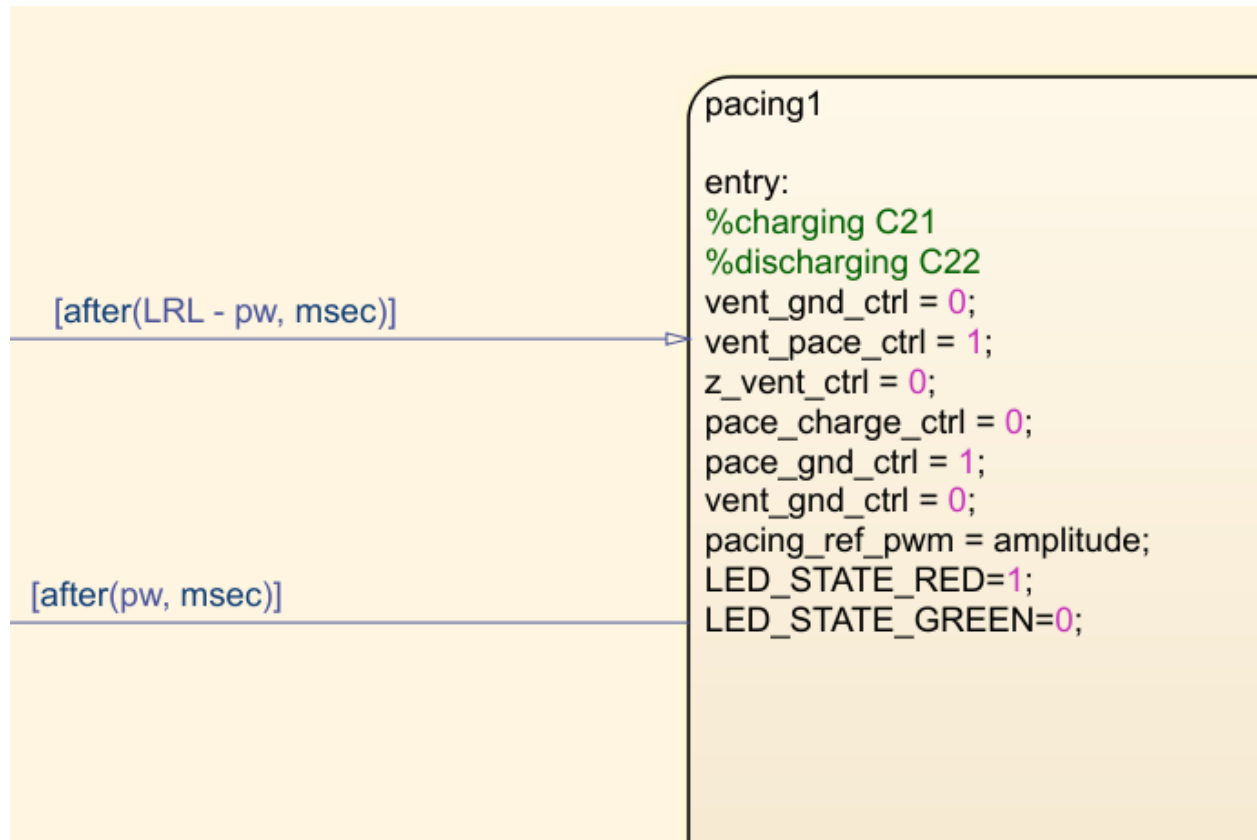


Figure 11: States' Edge Conditions and Pacing Conditions for the VOO

Similar to the AOO, the edge from charging to pacing state waits until the Lower Rate Limit - PW milliseconds as that will be that time until a pulse should be sent out as on cycle will be LRL + PW milliseconds. We then set the variable so that the Capacitor C21 is now the charging capacitor and Capacitor C22 is discharging which sends the signal that paces the heart. When the pacing state is reached, the red led is set to off and the green led is set to on



## States AAI & VVI

### AAI

**Brief Description:** This state listens to feedback from the atrium and will only send synthetic pulses if the heart is beating below the lower rate limit.

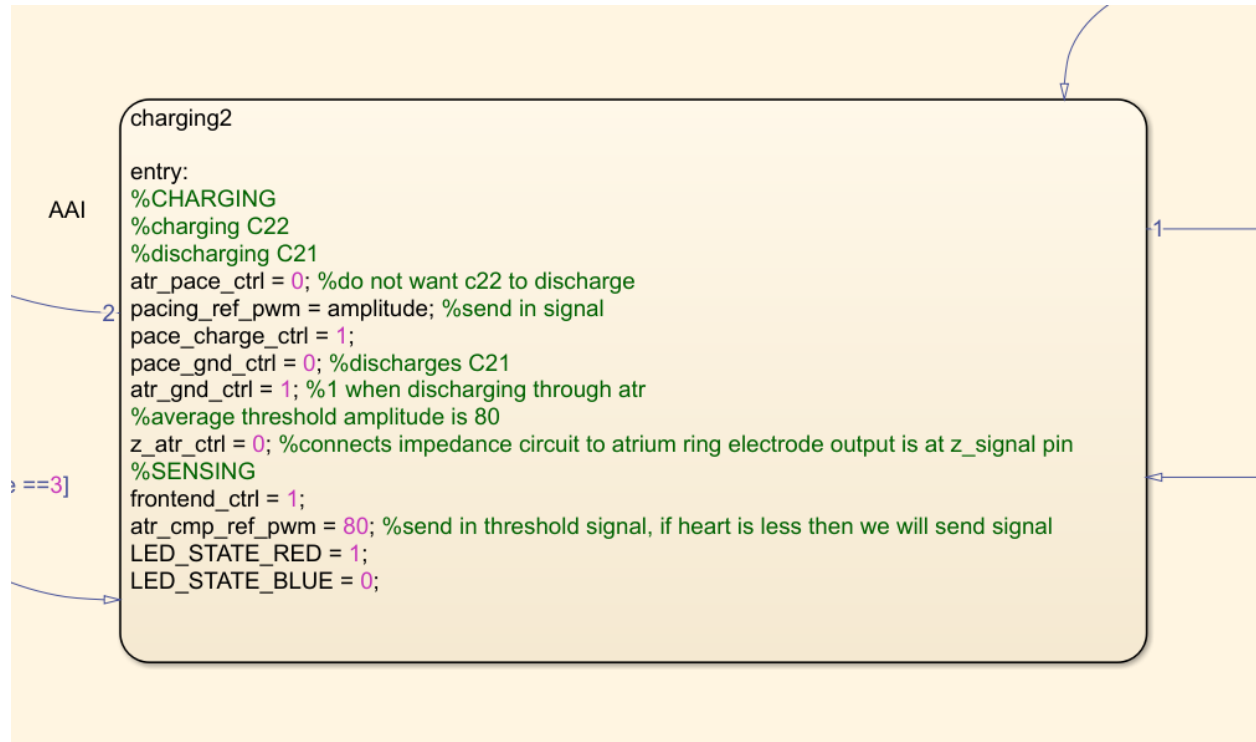


Figure 12: charging and sensing state AAI

Here, the state information is mostly the same in terms of charging state from AOO as the same C22 needs to be charged while C21 must be discharged. The only addition is that now we are sensing, and so there is digital input from atr\_cmp\_ref\_pwm which we use to check every 80 units of time whether or not we should be sending a signal, while LRL is the rate that we make sure the heart is beating in relation to.

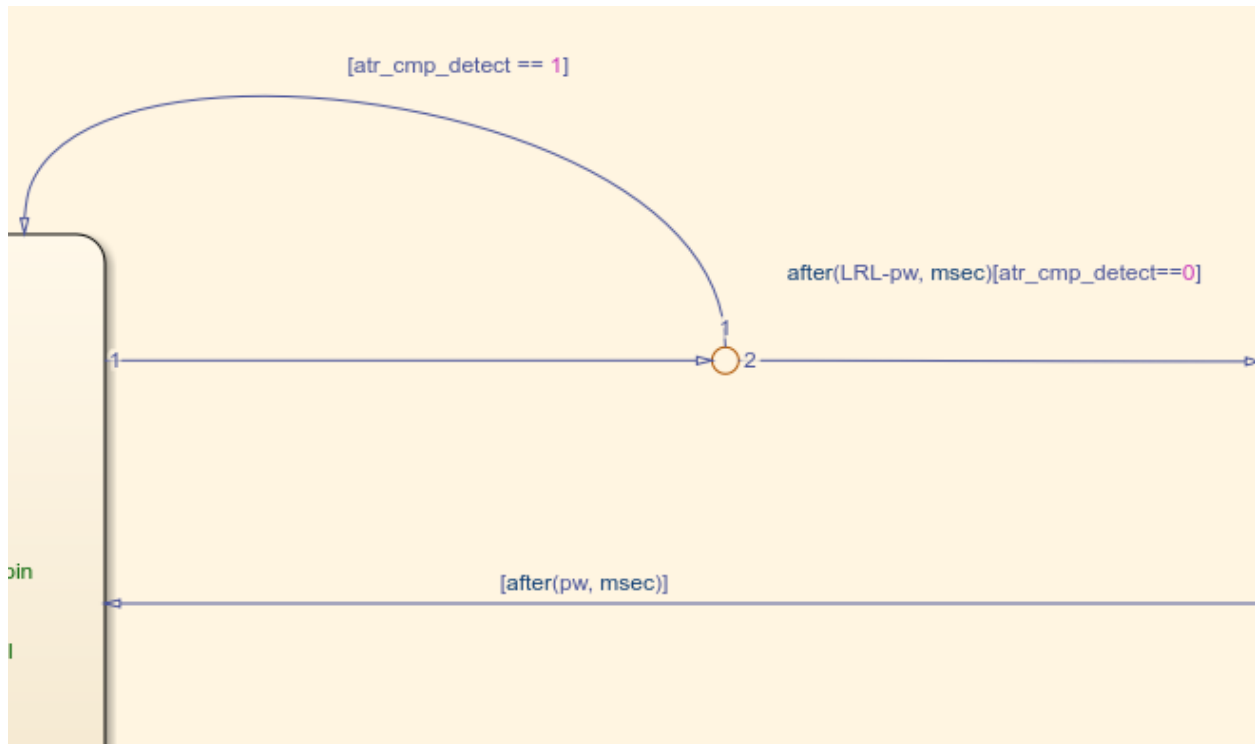


Figure 13: State transition and recursion call back to charging and sensing state

Here, the transition only occurs if atr\_cmp\_detect is 0 as this means that the rate is above the LRL.

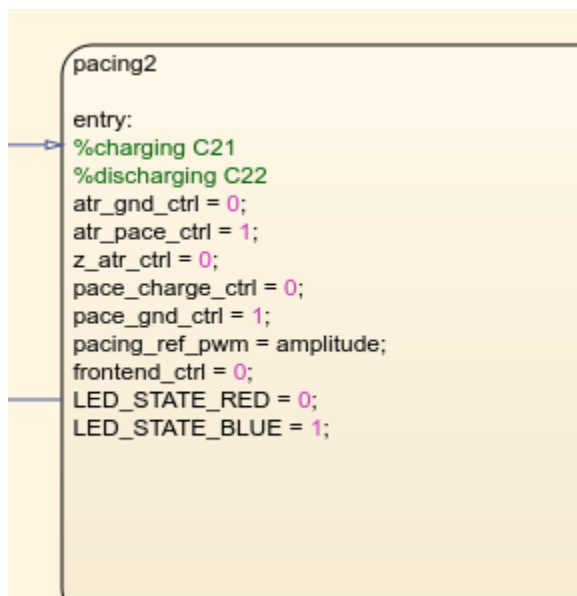


Figure 14: AAI Sensing state

This is the same pacing state as the AOO pacing state.

## VVI

The functionality of this state is the same as the functionality of AAI, and thus will not be explained.

**Brief Description:** This state listens to feedback from the ventricle and will only send synthetic pulses if the heart is beating below the lower rate limit.

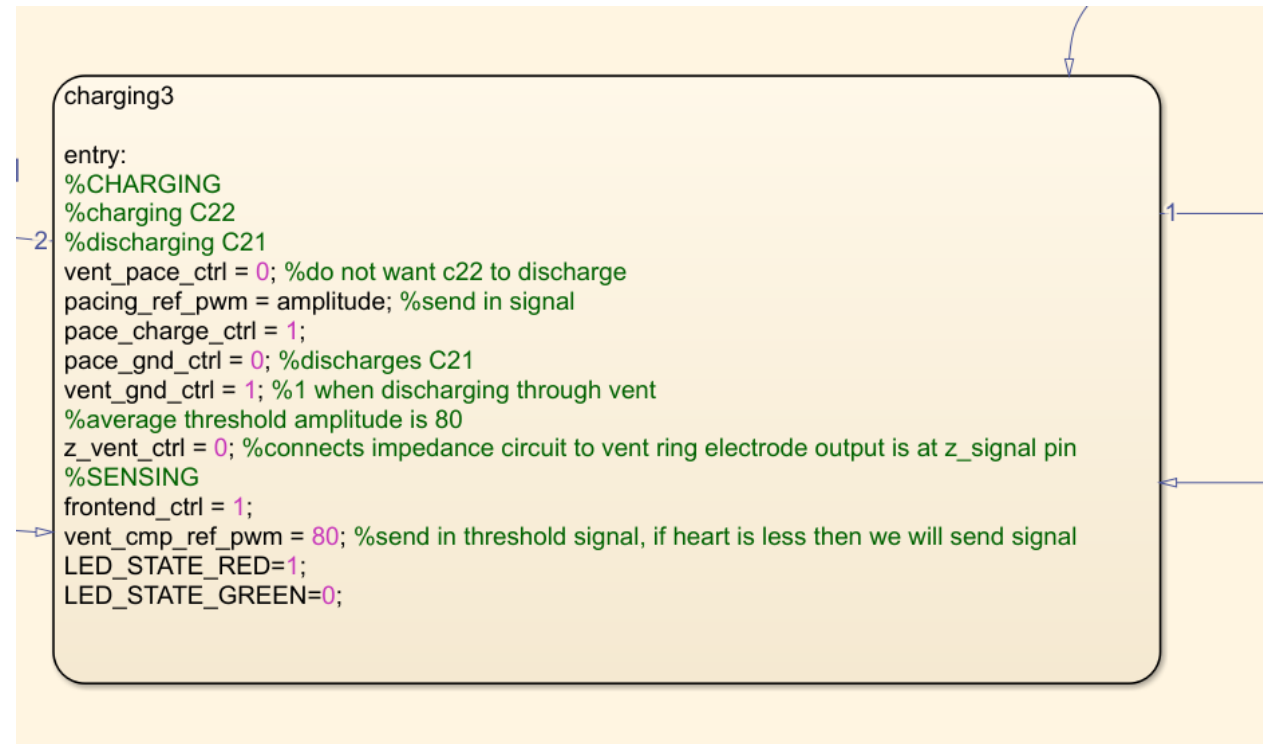


Figure 15: charging and sensing state VVI

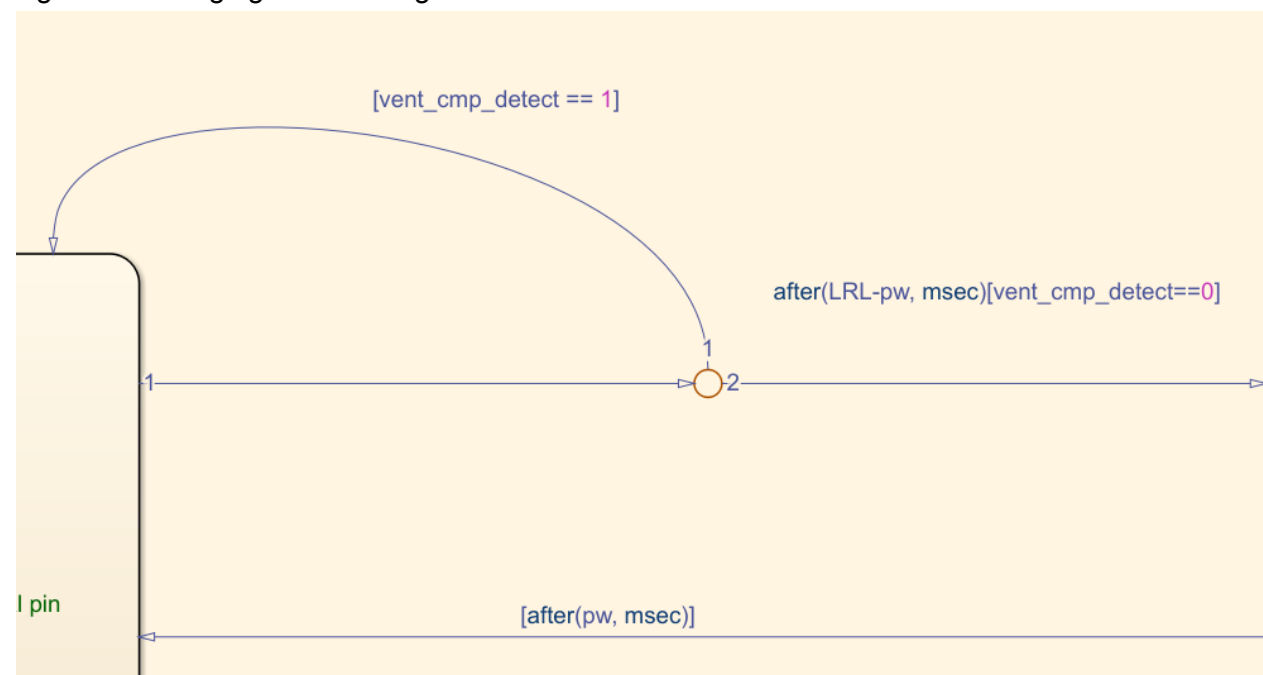


Figure 16: State transition and recursion call back to charging and sensing state

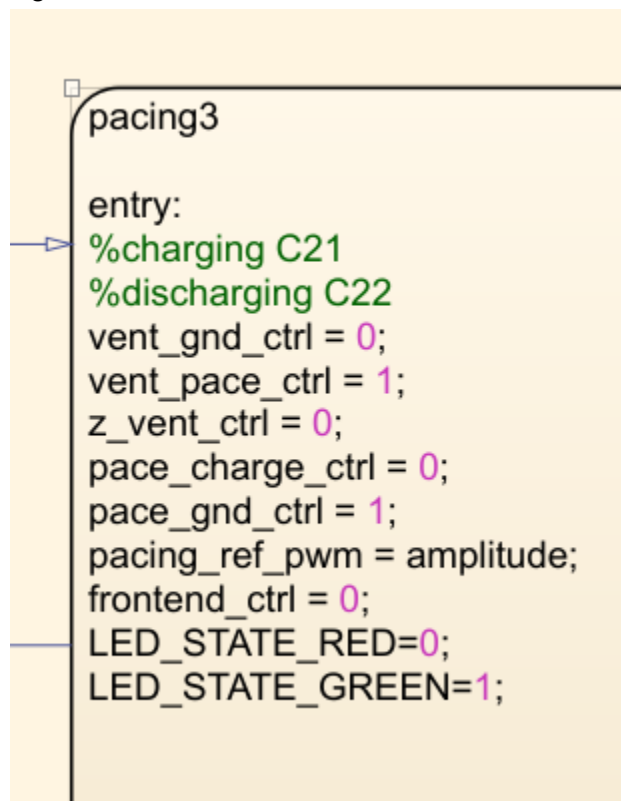


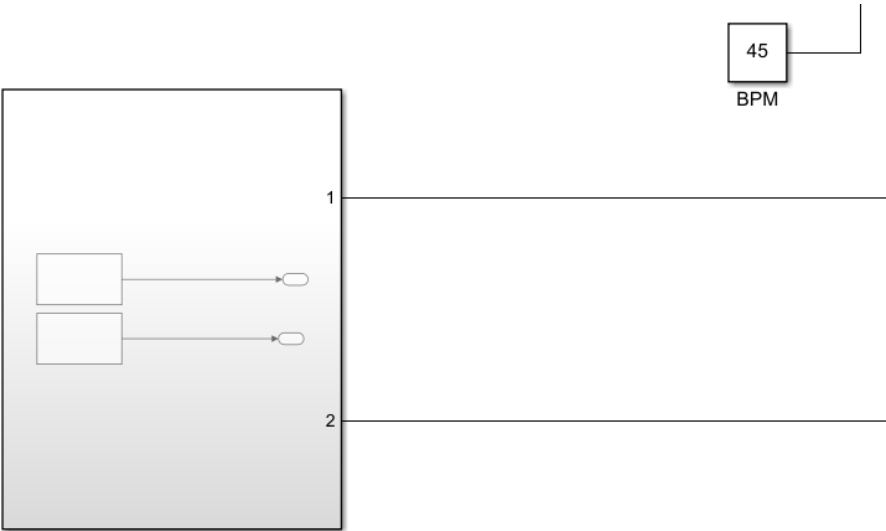
Figure 17: VVI pacing state

### Hardware Outputs

Parameter Name	Pin Assignment	Function
PACE_CHARGE_CTRL	D2	starts and stops the charging of the capacitor
Z_ATR_CTRL	D4	connects impedance circuit to atrium ring electrode output is at z_signal pin
VENT_CMP_REF_PWM	D3	send in threshold signal, if heart is less then we will send signal
PACING_REF_PWM	D5	Charges the capacitor of the pacing circuit
ATR_CMP_REF_PWM	D6	send in threshold signal, if heart is less then we will send signal
Z_VENT_CTRL	D7	connects impedance circuit to

		atrium ring electrode output is at z_signal pin
ATR_PACE_CTRL	D8	Discharges the primary capacitor through atrium
VENT_PACE_CTRL	D9	Discharges the primary capacitor through ventricle
PACE_GND_CTRL	D10	Allows current to flow from the ring to the tip
ATR_GND_CTRL	D11	Discharges the blocking capacitor through the atrium
VENT_GND_CTRL	D12	Discharges the blocking capacitor through the ventricle
FRONTEND_CTRL	D13	When ON displays the output to the heart signal

**Hardware Hiding**



**Sensing Subsystem that contains digital read inputs from pacemaker shield**

Figure 18: Hardware Hiding of inputs

### Hardware SubSystem Outputs: All outputs that go to pacemaker shield

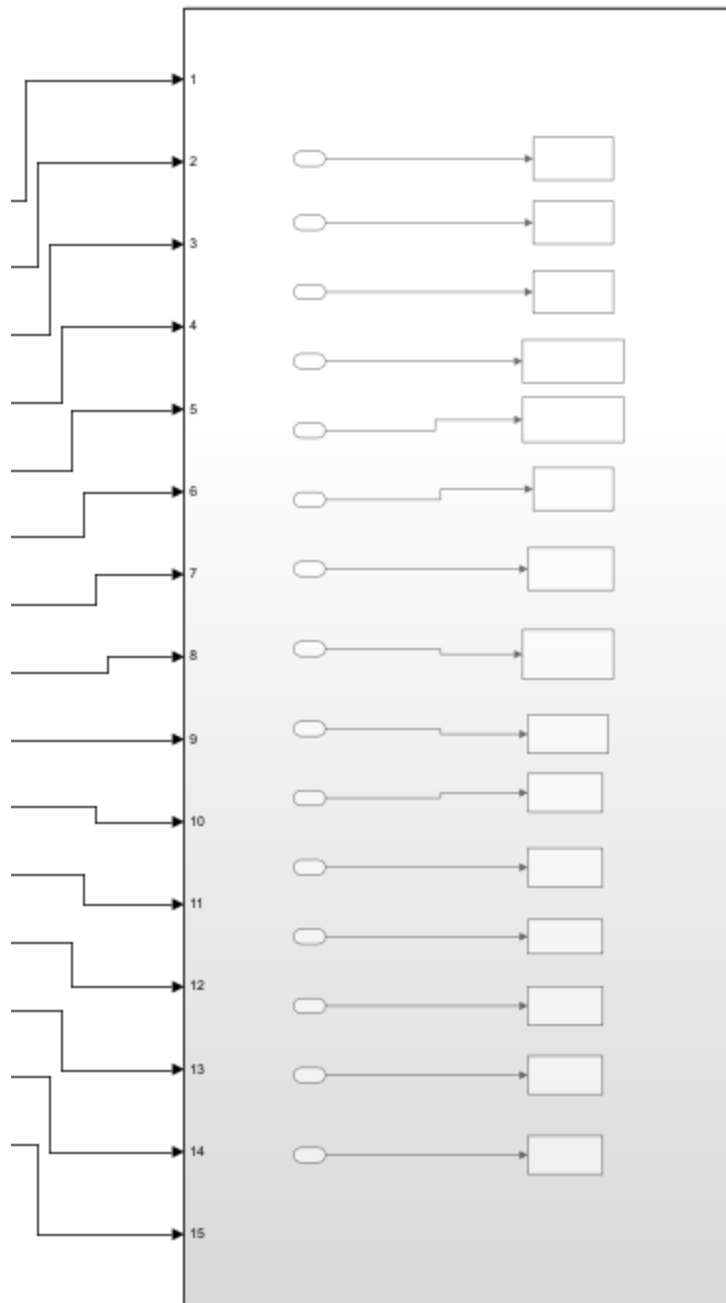


Figure 19: Hardware Hiding of outputs

Our system involves a level of hardware abstraction in the case that future additions to the project involve rearrangements of hardware pins. In the event that this happens, our model will not be affected due to our inclusion of subsystems. We used subsystems for the hardware inputs and outputs as can be seen in the two figures above. The names of the inputs and outputs to these subsystems were conveniently names identical to the pin names of the FRDM

board for the programmer's convenience but may be altered in the future to allow for an even deeper level of hardware abstraction.

- Inputs (input/parameter table), hardware inputs (pins), main stateflow (individual state descriptions), hardware outputs
- Simulink Individual State Descriptions include
- AOO & VOO
  - Charging and pacing
  - Explain the transitions
- AAI & VVI
  - Explain differences between these AOO and VOO

# DCM Module

## Introduction

We present an integrated software system consisting of three core files: `user.py`, `main.py`, and `registration.py`. These files work in harmony to provide a comprehensive user management, authentication, and registration solution.

'`user.py`' defines the `User` and `Accounts` classes, which encapsulate user account data and handle secure data management. '`main.py`' serves as the central control hub, orchestrating the application's core logic, state machine operations, user registration, and user authentication. It integrates the Tkinter library to create a visually engaging graphical user interface (GUI) that users can interact with. '`registration.py`' focuses on the registration process, ensuring the secure addition of users to the system.

This module's structure leverages the capabilities of `user.py`, `registration.py`, and `main.py` to deliver a robust and fully functional software system. `main.py` acts as the conductor, directing the overarching application behavior, managing the GUI, and facilitating user interactions, while `user.py` and `registration.py` provide the essential tools for user data management, security, and registration.

**IMPORTANT:** In order to run the GUI, you need to have the cryptography module downloaded  
*`pip3 install cryptography`*



## Authentication Module

The purpose of this authentication module is to handle user authentication and account management within a graphical user interface (GUI) using the tkinter library. It provides functions for logging in, signing up, and finding user information.

The logic regarding authentication is stored in the **registration.py** file.

All functions in the **registration.py** file are shown below:

Function Prototypes
find_user(users, username)
login(users, username, password)
signup(users, username, password)

find\_user(users, username)

Variable Name	Type	Description
users	User Object Array	Object array that stores the data of all locally signed up users.
username	String	The username we are trying to search

The '**find\_user**' function searches for the given 'username' within the 'users' object and returns the user object if found; otherwise, it returns -1.

The function works by iterating through the 'accounts' attribute of the 'users' object and returning the user object if a matching 'username' is found. If not found, it returns -1.

login(users, username, password)

Variable Name	Type	Description
users	User Object Array	Object array that stores the data of all locally signed up users.
username	String	The username that was entered by the user
password	String	The password that was entered by the user

The **'login'** function checks if the provided 'username' and 'password' match with an existing user. If the login is successful, it returns a tuple (1, this\_user) where 'this\_user' is the user object. If the login fails, it displays an error message and returns a tuple (0, this\_user).

The function calls find\_user to find the user with the provided 'username' and then verifies the 'password'. If successful, it returns a success bit and user information; otherwise, it shows an error message.

signup(users, username, password)

Variable Name	Type	Description
users	User Object Array	Object array that stores the data of all locally signed up users.
username	String	The username that was entered by the user
password	String	The password that was entered by the user

The **'signup'** function handles the signup process, ensuring that the 'username' is not already in use, there are fewer than 10 users, and that 'username' and 'password' are valid. If everything is correct, it will append a new user to the 'users object array' that will then be usable when logging in. Otherwise the function will throw an error with a message appropriate to the problem.

## User Module

The purpose of this file is to define two classes, "User" and "Accounts," which collectively manage user account data. It includes functionality to load and save user account information to/from files, create user accounts, and maintain user data in a secure manner using encryption. The "User" class represents individual user accounts, while the "Accounts" class manages these accounts and handles file I/O for data storage and retrieval. The module logic is stored onto the **user.py** file.

## User Class

The '**user**' class is used to represent a user with a name, password, and associated programmable data for each pace mode.

Class Variables	Type	Description
name	String	The username of the user
password	String	The password of the user
data	2D String Array	The programmable data of the user

## Class Constructor

Initializes a user object with a username, password and data.

## Accounts Class

The '**accounts**' class is used to manage user accounts, including reading from and writing to files, adding users, and loading data into memory.

Class Variables	Type	Description
accounts	List of User Objects	A list of User objects, representing user accounts.
length	Integer	The current number of user accounts.
old_serial	String	A string representing the old connected serial number of the hardware
serial	String	A string representing the current connected serial number of the hardware
accounts_file_path	String	The file path to the accounts data file.
device_file_path	String	The file path to the device data file.

All functions in the **Accounts** class in **user.py** file are shown below:

Class Function Prototypes
__init__(self)
load_accounts()
add_user(username, password)
update_device_file()
accounts_file_path
update_file()

## Class Constructor

The constructor initializes an Accounts object, including setting the paths for the accounts and device data files. It also calls `load_accounts()` to populate the `self.accounts` list with user data from the `accounts.txt` file and to retrieve the old serial from the `device.txt` file.

### `load_accounts()`

Function Used Variables	Type	Description
<code>self.accounts</code>	List of User Objects	A list of User objects, representing user accounts.
<code>self.old_serial</code>	String	A string representing the old connected serial number of the hardware
<code>self.accounts_file_path</code>	String	The file path to the accounts data file.
<code>self.device_file_path</code>	String	The file path to the device data file.

This function reads user account data from the accounts file, decrypts it using the `hardcoded_key`, and populates the `self.accounts` list with User objects. It also reads and stores the old serial from the device data file.

It first initializes the cryptography key cipher\_suite using the `hardcoded_key`. It opens the device data file (`self.device_file_path`) and reads the old serial, storing it in `self.old_serial`.

Then, it reads the accounts data from the `accounts.txt` file (`self.accounts_file_path`) line by line. For each line, it does the following: Decrypts the line's content using the encryption key. Parses the decrypted message, extracting the username, password, and associated data. Creates a new User object with this information and appends it to the `self.accounts` list. It finally increments the `self.length` counter.

`add_user(username, password)`

Function Used Variables	Type	Description
<code>self.accounts</code>	List of User Objects	A list of User objects, representing user accounts.
<code>self.length</code>	Integer	The current number of user accounts.
<code>self.accounts_file_path</code>	String	The file path to the accounts data file.

This function adds a new user to the `self.accounts` list if the number of user accounts is less than 10. It initializes the user with the user inputted username, password, nominal data and then calls `update_file()` to write the updated account information to the `accounts.txt` file.

`update_device_file()`

Function Used Variables	Type	Description
<code>self.serial</code>	String	A string representing the new connected serial number of the hardware
<code>self.device_file_path</code>	String	The file path to the device data file.

This function writes the current `self.serial` value to the `device.txt` data file.

update\_file()

Function Used Variables	Type	Description
self.accounts	List of User Objects	A list of User objects, representing user accounts.
self.accounts_file_path	String	The file path to the accounts data file.
hardcoded_key	String	A predefined encryption key.

This function will write the accounts data that is stored in memory to the accounts.txt file after encrypting it using the hardcoded\_key.

It iterates through the User objects in self.accounts and creates a string with the user's username, password and data. That string is then encrypted using the key and appended onto the accounts.txt file. This process is done for each user in the list.

## Main Module

### State Variables

Variable	Type	Description
users	User Object Array	A list of user objects with all usernames, passwords, and data
connected	Integer	1 when a device is connected, 0 when not connected
serial	String	The serial number of the currently connected device
AppState	Integer Enumeration	Keeps track of the current display (welcome or telemetry)
window	Object	The display window
frame	Object	The main frame for the window where all buttons, labels, and entries are shown

### clear\_frame()

Function Used Variables	Type	Description
widget	Object	Stores each widget on screen: buttons, labels, entries, etc.

This function clears every single widget on the current frame

### connection\_UI()

Function Used Variables	Type	Description
connected_label	Label Object	Label that displays whether a device is connected



This function displays whether a device is connected on screen, and if a device is connected, it says whether or not it's the same device as previously connected or a different one

`show_welcome_state()`

Function Used Variables	Type	Description
welcome_label	Label Object	Label to display welcome message
label_username	Label Object	Label to let user know where to type the username
entry_username	Entry Object	Entry box to type username
label_password	Label Object	Label to let user know where to type the password
entry_password	Entry Object	Entry box to type password
login_button	Button Object	Button to log in
signup_button	Button Object	Button to sign up

The **show\_welcome\_state** function displays the welcome window where the user can either log in or sign up. This function also calls the **connection\_UI** function to display whether a device is connected or not. This function also calls two other functions, **handle\_login** and **handle\_signup**, which I will talk about next.

`handle_login()`

Function Used Variables	Type	Description
username	String	Stores whatever the user enters in the username box
password	String	Stores whatever the user enters in the password box
login_check	Tuple of Bool and User Object	Stores whether the user logged in successful, and the user that was logged in if it was successful
current_user	User Object	Stores the current user that is

		logged in
--	--	-----------

This function calls the **login** function from **registration.py** to see if the login was successful. If it was successful, it stores the current user to be accessed later, then it updates the state to the telemetry state.

handle\_signup()

Function Used Variables	Type	Description
username	String	Stores whatever the user entered in the username box
password	String	Stores whatever the user entered in the password box

This function calls the **signup** function from **registration.py** to see if the signup was successful

show\_telemetry\_state()

Function Used Variables	Type	Description
modes	String Array	Array that stores the four mode names in order
params	String Array	Array that stores the eight parameter names in order
int_or_float	Integer Array	Array that stores whether each parameter is an integer or float (int = 1, float = 0)
params_increment	3D Float Array	Array that stores the information for each parameter with respect to the valid interval start, end, and increment
mode_settings	Dictionary(String : Integer Array)	Look-up table that tells the code which parameters to display based on the current mode

mode_order	Dictionary(String : Integer)	Look-up table that assigns a number/index to each mode
telemetry_label	Label Object	Label to display welcome message
selected	String	String that displays the current mode on the dropdown
dropdown	Dropdown Object	Dropdown menu to select mode
mode_button	Button Object	Button to select mode after selecting it in the dropdown
frame2	Frame Object	Frame to display telemetry label center justified

This function displays the telemetry screen with all the required information, including the dropdown, parameter labels, values, and entry boxes. This function also calls three other functions which I will talk about next.

update\_text()

Function Used Variables	Type	Description
entry_boxes	Entry Object Array	Array to hold all entry boxes on screen
error_labels	Label Object Array	Array to hold all parameter labels on screen
value_labels	Label Object Array	Array to hold all parameter value labels on screen
widget	Object	Stores all widgets on screen to be destroyed when cleared
mode_key	Integer Array	Holds the indices of all the parameters to be displayed in the current mode
current_mode	Integer	Holds the index of the current mode (AOO = 0, VOO = 1, etc)
mode_label	Label Object	Displays each parameter

		name on screen
value_label	Label Object	Displays each parameter value on screen
param_entry	Entry Object	Entry boxes where parameter values can be entered
error_label	Label Object	Displays error messages for each parameter if an invalid entry is entered
submit_button	Button Object	Button to submit changes

This method updates the parameter text on screen and displays any error messages if an invalid entry was entered. This function also calls **update\_params** when the submit button is pressed, which I will talk about next

update\_params(entries, key, errors, values, current\_mode)

Function Used Variables	Type	Description
entries	Entry Object Array	Same as entry_boxes from the previous function
key	Integer Array	Same as mode_key from the previous function
errors	Label Object Array	Same as error_labels from the previous function
values	Label Object Array	Same as value_labels from the previous function
current_mode	Integer	Same as current_mode from the previous function
entry_text	String	Gets what the user typed in the current parameter entry box

This functions checks if the entered text is valid, and then updates the parameter values on the display and the text file if the inputs are valid. It checks if the entries are the correct types, if the entries are not empty, and then it updates the text on screen if they are valid. When it's checking if the entries are valid (within the correct ranges), it calls the next function, **check\_input**

check\_input(label\_index, index, input\_data, errors)

Function Used Variables	Type	Description
label_index	Integer	The position of the entry box that is being checked
index	Integer	The index of the current parameter that is being checked
input_data	String	Holds what the user entered in the entry box
errors	Label Object Array	Same as errors from previous function
param_data	2D Array	List of valid intervals for that particular parameter
error_string	String	String that holds error message if the entry was invalid

Checks if the input is valid, based on if the entry is within the correct range, if it's on the right interval, etc. If the data is valid, it returns the value back to the previous function. If the data is invalid, it changes the corresponding error label to the error message from the **error\_string** variable.

update\_state(new\_state)

Function Used Variable	Type	Description
new_state	Enum Integer	Holds the state that you want to navigate to

This function changes the state that you're currently on

# Simulink Design Process

## Decisions

A major design decision is chosen so that the different modes (AOO,VOO,AAI, and VVI) will be in the same state. The state will perform the necessary actions that are required for these states. This design decision was chosen to allow the testing of each mode to occur easily without having to reupload the simulink file to the microcontroller constantly and be able to make necessary changes to the modes and their inputs easily in different states. This design decision has numerous other benefits which is that creating new modes for the pacemaker is easier as the mode would be added on to the state.

A major design decision for the 4 states (AOO,VOO,VVI,AOO) is that they would have 2 states for charging and pacing. The design decision made here is to combine the state for charging and discharging the blocking capacitor to one state called charging. As explained before, the charging state will both charge the blocking capacitor in order for pacing to occur and after pacing occurs, the blocking capacitor is first discharged by grounding it. The pacing state sends a pace to either the atrium or the ventricle. In the pacing state of the modes the pins listed have these values:

Pin Name	description
PACE_CHARGE_CTRL	LOW
PACE_GND_CTRL	HIGH
Z_ATR_CTRL	LOW
Z_VENT_CTRL	LOW
VENT_GND_CTRL (ATR_GND_CTRL)	LOW
VENT_PACE_CTRL (ATR_PACE_CTRL)	HIGH

Since the decision for charging and discharging the blocking capacitor has been sent to one state, the values of the pins would be listed as:

Pin Name	description
PACE_CHARGE_CTRL	HIGH
PACE_GND_CTRL	HIGH
Z_ATR_CTRL	LOW



## Errors

There were many decisions throughout the design process that were changed or scrapped in favor of other decisions that were deemed to fit better.

A decision that was scrapped in favor of another was the idea of having different state that represent the different modes and actions surrounding them. The different modes each had individual files. This made it hard to run and test code together since each file was separate and needed to be compiled every time a different mode needed to be tested. Due to this the design choice was scrapped in favor of what the final design was.

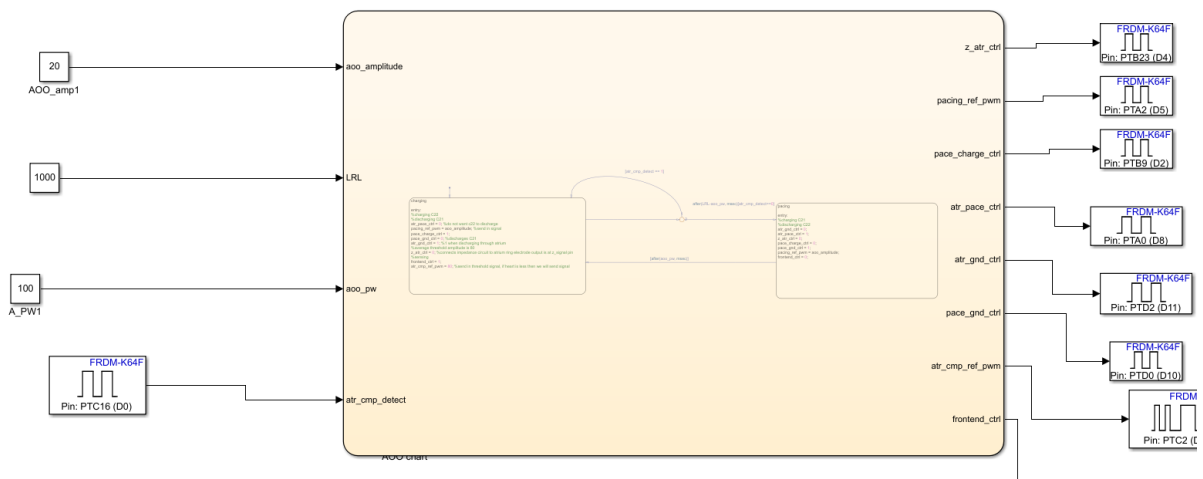


Figure 21 - AAO State table when separate from other states

The inputs and even the states in the design aren't changed from the final design as seen below. Both AOO and VOO both have a state for charging and discharging. The inputs for the lower rate limit and the pulse width are used as a threshold and duration for the electrical impulses. As seen, the different modes are designed the same as the final design except they are in different states. It made the transition to our final design simpler as there wasn't a need to change everything.

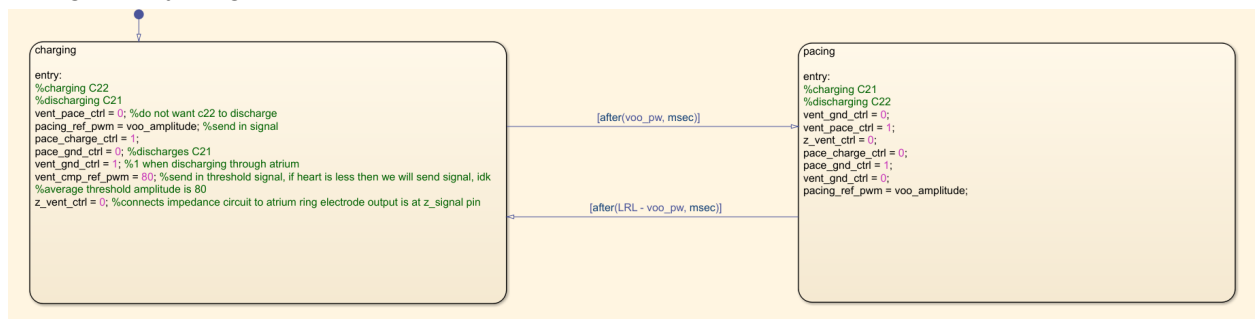


Figure 22 - AAO State table states

An idea that was deemed to be an error was when more requirements were being added to the model.



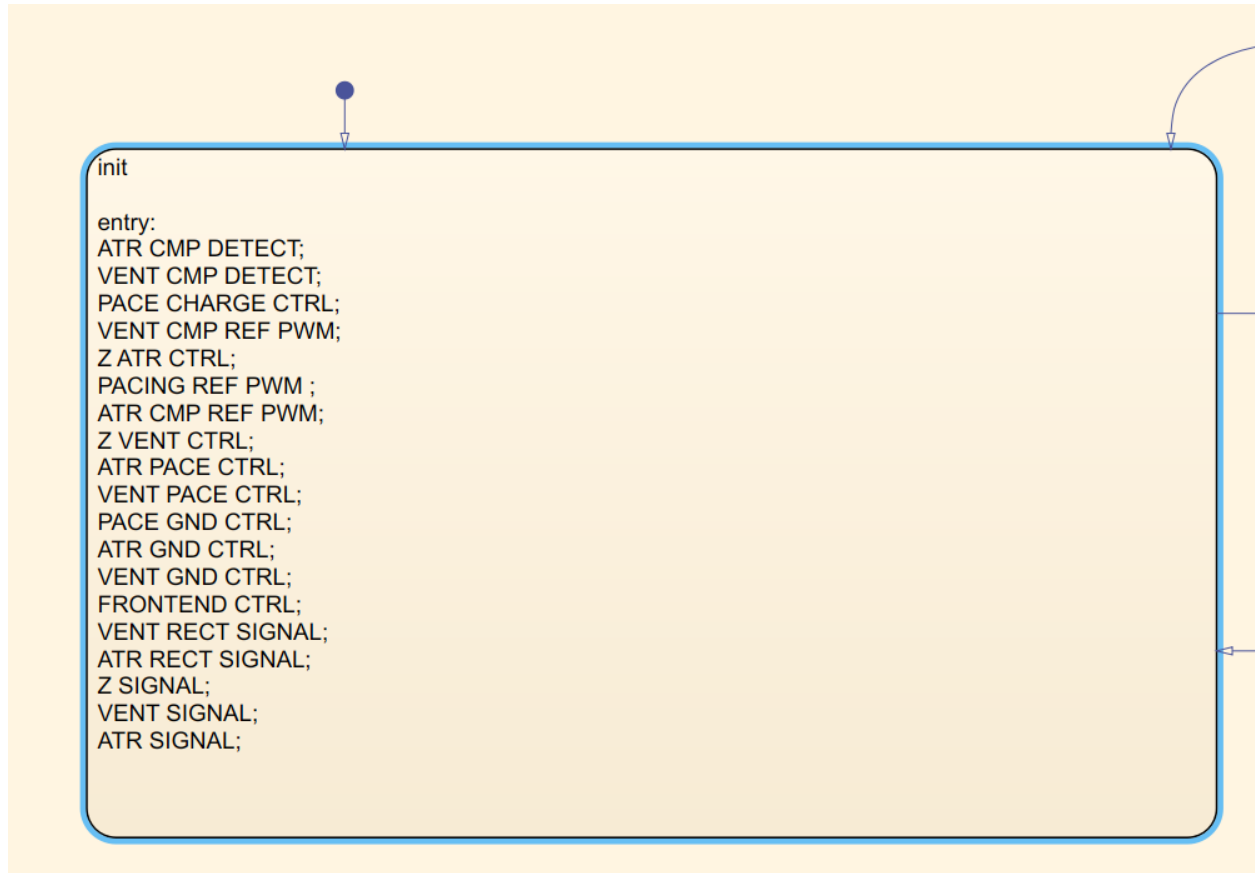


Figure 23 - List of all the old states added

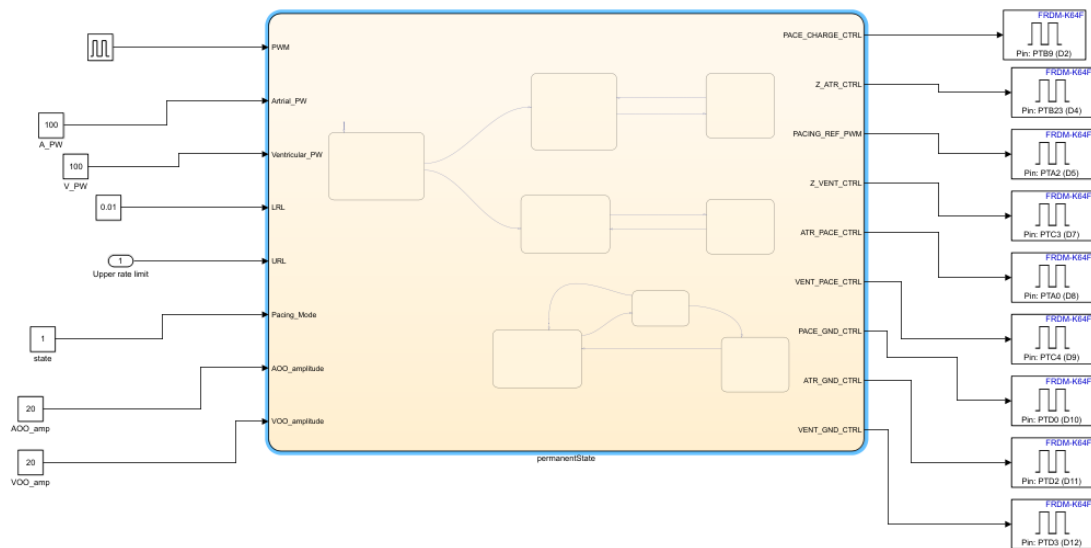


Figure 24 - Figure of all the inputs where a few were not needed

More states that do not have a function for the 4 modes for our requirements were being added. This meant that there were more inputs than there needed to be since most of these inputs did not serve a purpose to any of the modes. This error occurred due to misreading the requirements for each mode. The need for the input Upper rate limit ended up not being needed after the requirements were understood more carefully and the many states within the initialize function were removed right when work on the 4 modes was started as it was immediately known that there wouldn't be a need for some of the parameters when working on the modes: AOO, VOO,AAI, and VVI.

## Design Changes

A design decision that will be changed is where we have different states for the different modes except in the same file. The design will be changed to have them connected where an input changes them between modes. But this time the different modes will have different states as there will be more modes with different requirements and the simulink design will become messy if they all are placed into the same state. The different modes would be selected through a parameter where an input would represent different modes of the pacemaker. This allows for the different modes to be selected by the DCM in the future. As the requirements change to allow for more modes, there would need to be a way to be able to quickly change from different modes without much issues, which is where the parameter would come in.

Another design decision that we plan to have is instead of there being constant blocks for the inputs, the inputs will be determined by our DCM. The simulink model will receive the inputs from the DCM that will determine the input values for our different states and allow us to dynamically change the state. This will allow more freedom to not only test different states but help the pacemaker function automatically. The problems with the constant block that were being used is that despite being easy to test with different values, this does not accurately model the function of a pacemaker, as the inputs for the LRL, amplitude and pulse width will change depending on the different modes of the pacemaker. When these requirements are controlled by the DCM, it can allow for the system to be more personalized for each different user of the pacemaker.

Another design decision that is planned to change is to have subsystems for the inputs and outputs. This will allow the simulink design to be much more readable and neater. Having a neater design allows for newer inputs and outputs to be added without much difficulties. These subsystems can allow for information and data to be transferred to different states without difficulties.

# DCM Design Process

## Decisions

A major design process when creating the DCM was choosing a library to create the GUI. There were multiple options but we decided to go with Tkinter because it was wildly popular online and was built into python without the need of installing the module. At first, all the code was written into a singular file but that was an issue because it would make reading the code or expanding upon it very difficult. We decided to split the three main modules into their separate files (main, user and registration). For storing user data, we went back and forth between designs. We decided to go with storing data through a text file because it was very easy to read and write through python.

For storing the data for programmable inputs, we thought about making a json file that stored the min, max and intervals for each value but found it tedious to parse at run time. A simpler way we decided to go with is creating a 3D array which stores all the parameter's range, intervals in order to error check if the user inputs invalid numbers. This method makes it so all the data is already stored in RAM when the program is running, removing the initial step of parsing. A lot of decisions went into how we would store the user data and read it back. At this stage, we had the logic for writing onto the file but we were unsure on how to read it properly. We decided to make a function in the accounts class that would load all the data in the file to the computer's memory so we could use it.

Everything in the code was designed to be modular. We wanted to be able to easily expand on it during assignment 2 and the final assignment.

When changing pages, we were going to go with a big if statement but realized that this was very hard to read and not modular at all. We opted to go with a finite state machine. Initially the state machine will start in the welcome state where we can login/register and once logging in, the state changes to the next page (telemetry). When changing states it will call the respective function to show the new page. If we want to add more pages in the future, all we need to do is add a new state in the enum list and create the function that displays the page GUI. We have created a modular infrastructure so that once we add more programmable parameters and more modes it is just a matter of changing a singular list.

To hold the parameter range values, we also used a 3D array to store the interval ranges for each parameter.

## Errors

We had many roadblocks on the way. The first issue we came across was whenever you opened the project in a different folder, the program would no longer recognize the already

existing accounts.txt file. In order to fix this program, on run time, we get the location of the file and use it to call the file when reading/writing.

Another problem came when we were storing every programmable parameter differently for each mode. In the constructor, we would initialize the values in python using `[[list]]*4` in order to get the same list four times for each mode. Unfortunately this made it so we could not edit each mode's data specifically. We ended up fixing this bug by just writing the initial list properly without trying to take shortcuts `[[list],[list],[list],[list]]`.

Another problem was encryption. At first, we tried using hashlib in order to generate a random hash on the data. This worked where the data in accounts.txt was no longer readable but it made it so we could not decrypt it. We decided to opt with the cryptography library in python where given a key we were able to encrypt and decrypt.

## Design Changes

We have made many placeholders for assignment 1 where their full functionality will be utilized for upcoming assignments. For egrams, we will have a list which will store the output we read every 1-10 milliseconds in order from lowest index to highest so that we can later on graph the data and print it out as a report.

We have also made the logic for notifying the user once the hardware is connected, unconnected or if a NEW serial number is detected. Currently these are just variables controlling the logic but in the near future we will be reading the data from the hardware and updating the GUI on run time.

## Simulink Test Cases

### AOO Test Cases

The input parameters for the following 3 cases are as follows:

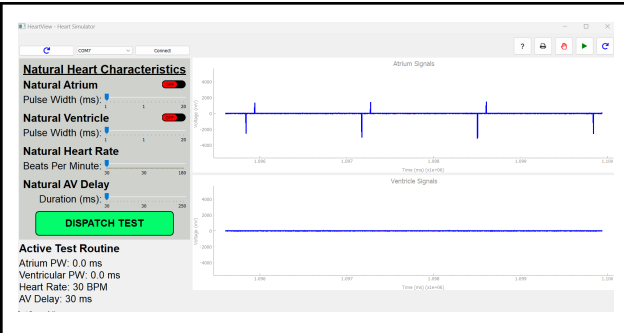
Amplitude = 20 mV

Pulse Width =  $100 * 10^6$  ms

#### Case 1: Pulsing With No Heart Rate

Expected Result	Actual Result
-----------------	---------------

Since there is no heart rate and this is AOO meaning that it does not sense any chambers, there should be no heart pulse but a synthetic pulse only.



Result: **Pass**

## Case 2: Pulsing With Small Heart Rate

Expected Result	Actual Result
Here the heart rate = 30 bpm, which is very low for the average adult. The expected result is that the pulses should occur at a constant pulse width relative to each other, regardless of the refractory period.	<p>The screenshot shows the HeartView - Heart Simulator interface. On the left, the 'Natural Heart Characteristics' section includes sliders for 'Natural Atrium Pulse Width (ms)' (set to 1), 'Natural Ventricle Pulse Width (ms)' (set to 1), 'Natural Heart Rate' (set to 30 BPM), and 'Natural AV Delay' (set to 30 ms). Below these is a 'DISPATCH TEST' button. The 'Active Test Routine' section shows 'Atrium PW: 1.0 ms', 'Ventricular PW: 0.0 ms', 'Heart Rate: 30 BPM', and 'AV Delay: 30 ms'. On the right, the 'Atrium Signals' plot shows a series of sharp, narrow pulses, while the 'Ventricle Signals' plot shows a flat line, indicating no ventricular activity.</p>

Result: **Pass**

## Case 3: Pulsing With Large Heart Rate

Expected Result	Actual Result
Here the heart rate = 87 bpm, which is very high for the average adult. The expected result is that the pulses should occur at a constant pulse width relative to each other, regardless of the refractory period.	<p>The screenshot shows the HeartView - Heart Simulator interface. On the left, the 'Natural Heart Characteristics' section includes sliders for 'Natural Atrium Pulse Width (ms)' (set to 1), 'Natural Ventricle Pulse Width (ms)' (set to 1), 'Natural Heart Rate' (set to 87 BPM), and 'Natural AV Delay' (set to 30 ms). Below these is a 'DISPATCH TEST' button. The 'Active Test Routine' section shows 'Atrium PW: 1.0 ms', 'Ventricular PW: 0.0 ms', 'Heart Rate: 87 BPM', and 'AV Delay: 30 ms'. On the right, the 'Atrium Signals' plot shows a series of sharp, narrow pulses, while the 'Ventricle Signals' plot shows a flat line, indicating no ventricular activity.</p>

Result: **Pass**

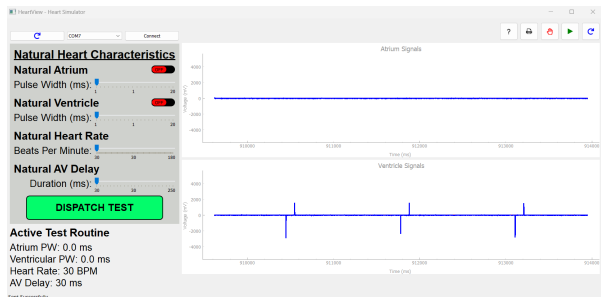
## VOO Test Cases

The input parameters for the following 3 cases are as follows:

Amplitude = 20 mV

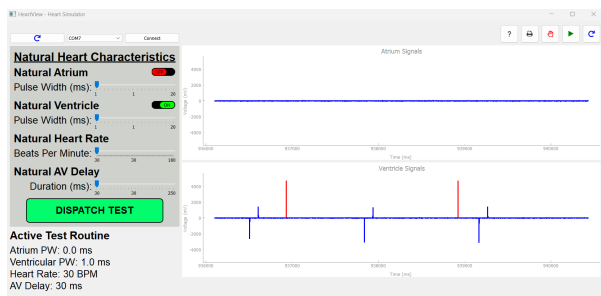
Pulse Width =  $100 \times 10^{-6}$  ms

## Case 1: Pulsing With No Heart Rate

Expected Result	Actual Result
<p>Since there is no heart rate and this is VOO meaning that it does not sense any chambers, there should be no heart pulse but a synthetic pulse only.</p>	

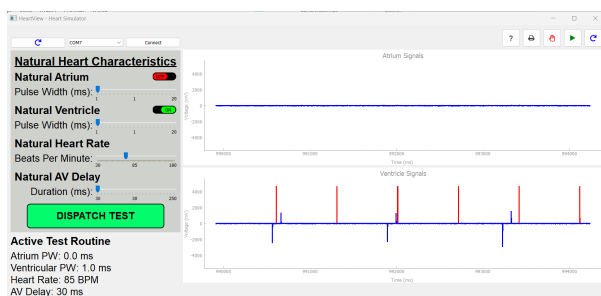
Result: **Pass**

## Case 2: Pulsing With Small Heart Rate

Expected Result	Actual Result
<p>Here the heart rate = 30 bpm, which is very low for the average adult. Since VOO doesn't listen for feedback, the expected result is that the pulses should occur at a constant pulse width relative to each other, regardless of the refractory period.</p>	

Result: **Pass**

## Case 3: Pulsing With Large Heart Rate

Expected Result	Actual Result
<p>Here the heart rate = 85 bpm, which is very high for the average adult. The expected result is that the pulses should occur at a constant pulse width relative to each other, regardless of the refractory period.</p>	

Result: **Pass**

## AAI Test Cases

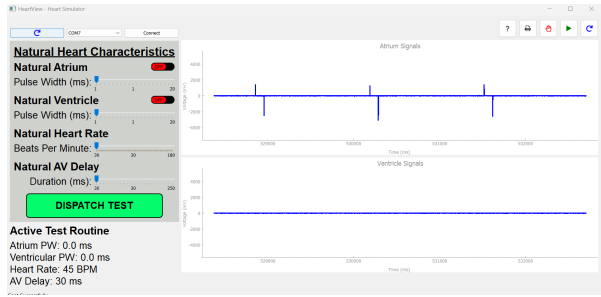
The input parameters for the following 3 cases are as follows:

Amplitude = 20 mV

Pulse Width =  $100 \times 10^{-6}$  ms

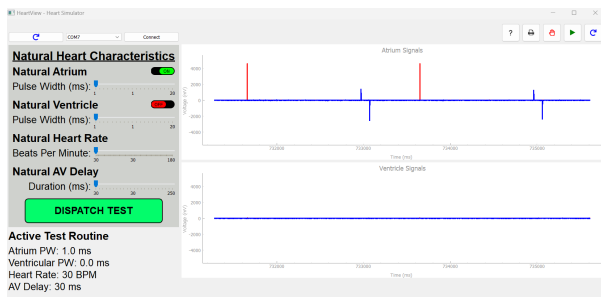
LRL = 45 bpm

### Case 1: Pulsing With No Heart Rate

Expected Result	Actual Result
<p>Since this is AAI, meaning the Atrium is being sensed and the response is inhibited depending on the heart, the expect result should be that there is only a synthetic pulse. This is due to the fact that the heart rate falls below the Lower rate limit.</p>	

Result: **Pass**

### Case 2: Pulsing With Heart Rate < LRL

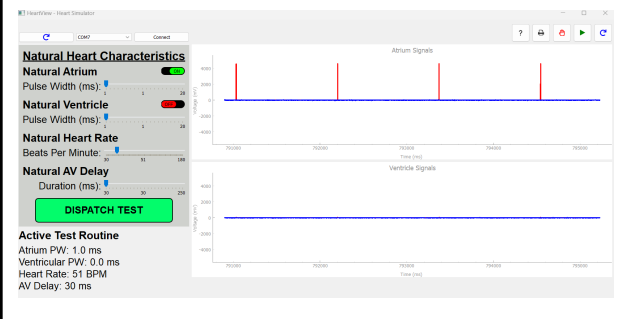
Expected Result	Actual Result
<p>Similar to the case above, since we are still beneath the LRL the synthetic pulse should be sent.</p>	

Result: **Pass**

### Case 3: Pulsing With Heart Rate > LRL

Expected Result	Actual Result
-----------------	---------------

Now that the heart rate is higher than the LRL the heart is fine and thus beats independently. No synthetic pulse should be detected.



**Result: Pass**

## VVI Test Cases

The input parameters for the following 3 cases are as follows:

Amplitude = 20 mV

Pulse Width =  $100 \times 10^6$  ms

LRL = 45 bpm

### Case 1: Pulsing With No Heart Rate

Expected Result	Actual Result
These cases are similar to the ones of AAI. Since it is sensing the ventricle and inhibiting its response based on the heart beat, there should only be a synthetic pulse.	<p>The screenshot shows the 'HeartView - Heart Simulator' window for Case 1. The 'Natural Heart Characteristics' section has 'Natural Atrium' and 'Natural Ventricle' pulse widths set to 1.0 ms. The 'Natural Heart Rate' is set to 30 BPM, and the 'Natural AV Delay' is 30 ms. The 'DISPATCH TEST' button is green. The 'Active Test Routine' section shows 'Atrium PW: 0.0 ms', 'Ventricular PW: 0.0 ms', 'Heart Rate: 30 BPM', and 'AV Delay: 30 ms'. The signal plots show 'Atrium Signals' as a flat blue line and 'Ventricle Signals' as a series of red vertical pulses, indicating that only synthetic ventricular pulses are present.</p>

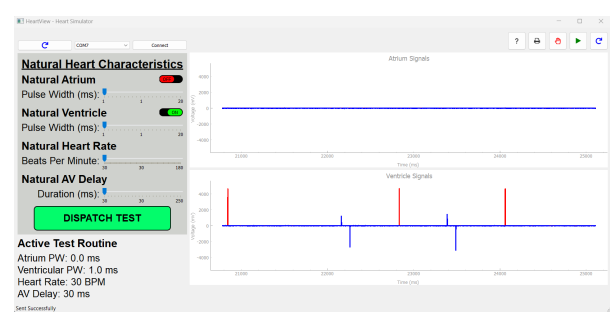
**Result: Pass**

### Case 2: Pulsing With Heart Rate < LRL

Expected Result	Actual Result
-----------------	---------------

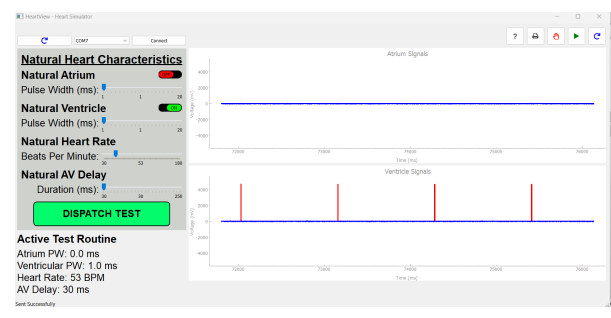


Since the heart rate is lower than LRL, there should only be a synthetic pulse.



Result: **Pass**

### Case 3: Pulsing With Heart Rate > LRL

Expected Result	Actual Result
Now that the heart rate is higher than the LRL, which we set to be 45 bpm as a programmable parameter, there should be strictly a synthetic pulse.	 <p>The screenshot shows the 'HeartView - Heart Simulator' window. On the left, under 'Natural Heart Characteristics', the 'Natural Heart Rate' is set to 53 BPM. The 'Active Test Routine' section shows 'Heart Rate: 53 BPM'. On the right, there are two signal plots: 'Atrium Signals' and 'Ventricle Signals'. The 'Ventricle Signals' plot shows a synthetic pulse, while the 'Atrium Signals' plot is flat at zero.</p>

Result: **Pass**

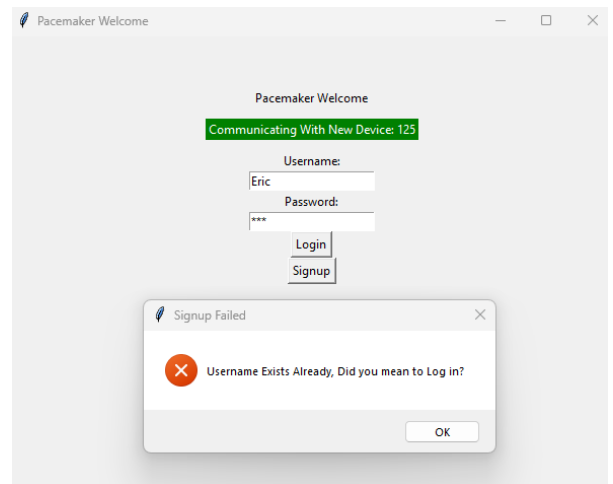
## DCM Test Cases

### Sign-in Page

#### Case 1: Signing Up with Pre-existing User

Expected Result	Actual Result
-----------------	---------------

Since the user “Eric” already exists, signing up with his user should give an error.



Result: **Pass**

### Case 2: Signing In with Non-Existent User

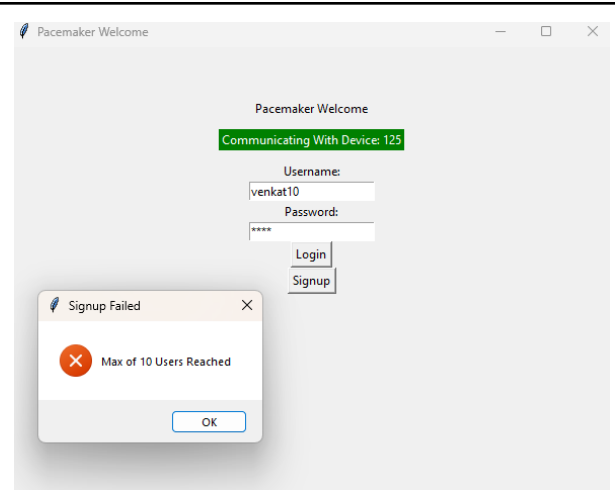
Expected Result	Actual Result
Since the user “Michael” doesn’t exist, it should return an error	A screenshot of the "Pacemaker Welcome" application. The form shows "Username:" as "Michael" and "Password:" as three asterisks. The "Login" button is visible. The green status bar at the top says "Communicating With Device: 125". A "Login Failed" error dialog is displayed, featuring a red 'X' icon and the text "Incorrect username or password". An "OK" button is located at the bottom of the dialog.

Result: **Pass**

### Case 3: User Number Limit of 10

Expected Result	Actual Result
-----------------	---------------

Since there is the main user “Eric” already registered, the 10th “Venkat” user should not be able to be made and it should return an error



Result: **Pass**

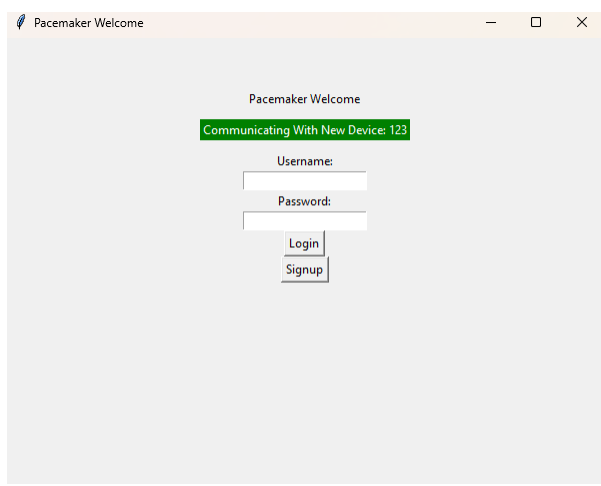
#### Case 4: Username Has Spaces

Expected Result	Actual Result
Username being signed up with spaces should return an error	

Result: **Pass**

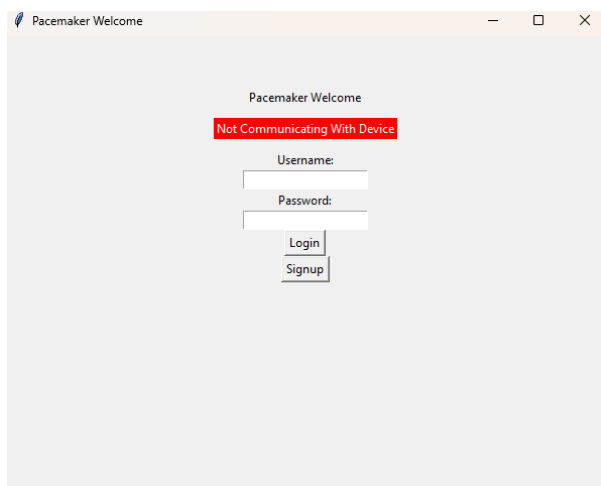
#### Case 5: Changing Serial Number

Expected Result	Actual Result
-----------------	---------------

<p>Since the serial number is changed from the classic 125 to 123, it should display that it's from a new hardware device.</p>	 <p>The screenshot shows a web browser window titled 'Pacemaker Welcome'. The page content includes the title 'Pacemaker Welcome', a green status box with the text 'Communicating With New Device: 123', and a login form with fields for 'Username:' and 'Password:', and buttons for 'Login' and 'Signup'.</p>
--	---

Result: **Pass**

### Case 6: Changing Connection

Expected Result	Actual Result
<p>Since the connection flag has been turned from 1 to 0, it should give an error for the communication device</p>	 <p>The screenshot shows a web browser window titled 'Pacemaker Welcome'. The page content includes the title 'Pacemaker Welcome', a red status box with the text 'Not Communicating With Device', and a login form with fields for 'Username:' and 'Password:', and buttons for 'Login' and 'Signup'.</p>

Result: **Pass**

## Telemetry Page

### Case 1: Changing State

Expected Result	Actual Result
-----------------	---------------

Since the user “Eric” has changed the state from AOO to VOO, even after exiting the program and rebooting it the information of what was submitted in the new selection should be saved

Welcome to Telemetry

Communicating With Device: 125

VOO

Select

Lower Rate Limit: 60 65

Upper Rate Limit: 120 125

Ventricular Amplitude: 3.5 4

Ventricular Pulse Width: 0.2 0.3

Submit Changes

Telemetry

Welcome to Telemetry

Communicating With Device: 125

VOO

Select

Lower Rate Limit: 65

Upper Rate Limit: 125

Ventricular Amplitude: 4.0

Ventricular Pulse Width: 0.3

Submit Changes

Result: **Pass**

## Case 2: Testing Values Not Following Increment

Expected Result	Actual Result
-----------------	---------------

Since AOO only takes increments of 5 between 90 and 175 for lower rate limit, this should give an error to an input of 91

The screenshot shows a web application titled "Telemetry". At the top, it says "Welcome to Telemetry" and "Communicating With Device: 125". Below this, there are two buttons: "AOO" and "Select". The main configuration area has four rows of labels and input fields:

- Lower Rate Limit: 90 | 91
- Upper Rate Limit: 125 | 125
- Ventricular Amplitude: 4 | 4
- Ventricular Pulse Width: 0.3 | 0.3

Below the input fields is a "Submit Changes" button. To the right of the "Lower Rate Limit" input field, there is a red error message: "Between 90 and 175, increment must be 5".

Result: **Pass**

**Case 3: State Change Mid-Value change**

Expected Result	Actual Result
-----------------	---------------

Since the changing of dropdown should only happen when select is clicked, adding a new value for a state, switching states in the dropdown, submitting changes and then pressing select should only make the changes to the previous state and display the new state

Telemetry

Welcome to Telemetry

Communicating With Device: 125

VVI

Lower Rate Limit: 65

Upper Rate Limit: 120

Ventricular Amplitude: 3.5

Ventricular Pulse Width: 0.4

VRP: 500

Telemetry

Welcome to Telemetry

Communicating With Device: 125

AAI

Lower Rate Limit: 65

Upper Rate Limit: 120

Ventricular Amplitude: 3.5

Ventricular Pulse Width: 0.4

VRP: 500

Telemetry

Welcome to Telemetry

Communicating With Device: 125

AAI

Lower Rate Limit: 65

Upper Rate Limit: 120

Ventricular Amplitude: 3.5

Ventricular Pulse Width: 0.4

VRP: 400

Telemetry

Welcome to Telemetry

Communicating With Device: 125

AAI

Lower Rate Limit: 60

Upper Rate Limit: 120

Atrial Amplitude: 3.5

Atrial Pulse Width: 0.4

ARP: 200

Result: **Pass**

Case 4: User Independence

Expected Result	Actual Result
After two users are made, changes in one users information should not change that information for the other user	<div><div><div><div><div><div>Pacemaker Welcome</div><div>Communicating With Device: 125</div><div><div>Username: Venkat1</div><div><div>Password: ***</div><div><div>Login</div><div>Signup</div></div></div><div><div>Signup Successful</div><div>Account created for Venkat1!</div><div>OK</div></div></div></div></div></div><div><div><div><div><div>Welcome to Telemetry</div><div>Communicating With Device: 125</div><div><div>VOO Select</div><div><div>Lower Rate Limit: 60</div><div></div></div><div><div>Upper Rate Limit: 120</div><div></div></div><div><div>Ventricular Amplitude: 3.5</div><div></div></div><div><div>Ventricular Pulse Width: 0.4</div><div></div></div><div>Submit Changes</div></div></div></div></div><div><div><div><div><div>Welcome to Telemetry</div><div>Communicating With Device: 125</div><div><div>VOO Select</div><div><div>Lower Rate Limit: 65</div><div>65</div></div><div><div>Upper Rate Limit: 125</div><div>125</div></div><div><div>Ventricular Amplitude: 4</div><div>4</div></div><div><div>Ventricular Pulse Width: 0.5</div><div>0.5</div></div><div>Submit Changes</div></div></div></div></div><div><div><div><div><div>Welcome to Telemetry</div><div>Communicating With Device: 125</div><div><div>VOO Select</div><div><div>Lower Rate Limit: 60</div><div></div></div><div><div>Upper Rate Limit: 120</div><div></div></div><div><div>Ventricular Amplitude: 3.5</div><div></div></div><div><div>Ventricular Pulse Width: 0.4</div><div></div></div><div>Submit Changes</div></div></div></div></div></div></div></div></div></div>

Result: **Pass**



# Conclusion

This documentation provides a comprehensive overview of the Simulink and DCM functionalities, emphasizing the development of the four essential modes, the welcome screen, and their corresponding requirements and operational principles. The modes are explored in depth, and the significance of key parameters is thoroughly elaborated upon, shedding light on how both the DCM and Simulink can operate effectively with needed changes.

The discussion on requirements serves as a foundation for understanding future changes in the modes to be handled and the evolving capabilities of the DCM, particularly addressing the need for additional states to accommodate newer modes. The planned integration of serial communication is poised to empower the DCM to send and receive information from the pacemaker, thereby enhancing control and data retrieval. In the coming stages of the project, we anticipate achieving full communication with the pacemaker, which will grant access to egrams data and facilitate the development of user reports.

This enhancement is crucial for optimizing the pacemaker's functionality in response to user input. This documentation, thus, serves as a pivotal resource for comprehending the pivotal components and ever-evolving requirements of the pacemaker project. It not only underlines the core functionalities and the intricacies of the Simulink and DCM systems but also outlines the roadmap for future enhancements, ensuring that the pacemaker system remains adaptable and responsive to the needs of both patients and medical practitioners.