Pacemaker Project Documentation

Group 32

Logan Eby,

Prawin Premachandran,

Ahmad Molhim,

Riley Chai,

Youssef El Ashmawy,

Andrew DePetris

Table of Contents

List of Figures

List of Tables

1 Requirements

1.1 Current Requirements

1.1.1 Simulink

For current requirements, the pacemaker software is required to activate the AOO, VOO, AAI, VVI, AOOR, VOOR, AAIR and VVIR modes based on user input and parameters listed below. The requirements for the individual modes are detailed in the following sections.

Parameter	Programmable Values	Increment	Nominal	Tolerance
Lower Rate Limit	30-50 ppm 50-90 ppm 90-175 ppm	5 ppm 1 ppm 5 ppm	60 ppm	±8 ms
Upper Rate Limit	50-175 ppm	5 ppm	120 ppm	±8 ms
A or V Amplitude	Off, 0.5-3.2V 3.5-7.0 V	0.1V 0.5V	3.75 V	—
A or V Pulse Width	0.05 ms 0.1-1.9 ms	— 0.1ms	0.4 ms	0.2 ms
Hysteresis Rate Limit	30-50 ppm 50-90 ppm 90-175 ppm	—	Off	±8 ms
VRP	150-500 ms	10 ms	320 ms	±8 ms
ARP	150-500 ms	10 ms	250 ms	±8 ms
MSR	50-175 ms	5 ms	120 ppm	±5 ms
Activity Threshold	V-Low, Low, Med-Low, Med, Med-High, High, V-High		Med	
Reaction Time	10-50 s	10 s	30 s	±3 s
Response Factor	1-16	1	8	
Recovery Time	2-16 min	1 min	5 min	±30 s

Table 1: Pacemaker Parameters and range of expected values

1.1.1.1 AOO Requirements

The AOO mode paces the atrium at a fixed, defined interval independent of any natural atrium heart activity. The fixed interval used is the Lower Rate Limit. The pulse is determined by the Atrial Pulse Width, defined by the user above.

Given: The user sets the pacemaker to AOO and inputs the required parameters

When: Lower Rate Limit has passed

Then: The pacemaker paces the atrium for an Atrial Pulse Width pulse

1.1.1.2 VOO Requirements

The VOO mode paces the ventricle at a fixed, defined interval independent of any natural ventricle heart activity. The fixed interval used is the Lower Rate Limit. The pulse is determined by the Ventricle Pulse Width, defined by the user above.

Given: The user sets the pacemaker to VOO and inputs the required parameters

When: Lower Rate Limit has passed

Then: The pacemaker paces the ventricle for a Ventricle Pulse Width pulse

1.1.1.3 AAI Requirements

The AAI mode paces the atrium at a fixed, defined interval so long as there is no natural atrium heart activity. The fixed interval used is the Lower Rate Limit. The pulse is determined by the Atrial Pulse Width, defined by the user above. If there is natural atrium heart activity between paces and after the Atrial Refraction Period, the AAI mode does not pace for an interval defined by Lower Rate Limit + Hysteresis Interval. If there is natural atrium heart activity in this interval, the interval resets.

Given: The user sets the pacemaker to AAI and inputs the required parameters

When: Lower Rate Limit has passed and there has been no natural atrium heart activity, or the atrium heart activity took place in the Atrial Refraction Period

Then: The pacemaker paces the atrium for an Atrial Pulse Width pulse

Given: The user sets the pacemaker to AAI and inputs the required parameters

When: There has been natural atrium heart activity between paces and after the Atrial Refraction Period

Then: The pacemaker does not pace for an interval defined by Lower Rate Limit + Hysteresis Interval

1.1.1.4 VVI Requirements

The VVI mode paces the atrium at a fixed, defined interval so long as there is no natural ventricle heart activity. The fixed interval used is the Lower Rate Limit. The pulse is determined by the Ventricle Pulse Width, defined by the user above. If there is natural ventricle heart activity between paces and after the Ventricle Refraction Period, the VVI mode does not pace for an interval defined by Lower Rate Limit + Hysteresis Interval. If there is natural ventricle heart activity in this interval, the interval resets and there is still no pulse.

Given: The user sets the pacemaker to VVI and inputs the required parameters

When: Lower Rate Limit has passed and there has been no natural ventricle heart activity, or the ventricle heart activity took place in the Ventricle Refraction Period

Then: The pacemaker paces the atrium for a Ventricle Pulse width pulse

Given: The user sets the pacemaker to VVI and inputs the required parameters

When: There has been natural ventricle heart activity between paces and after the Ventricle Refraction Period

Then: The pacemaker does not pace for an interval defined by Lower Rate Limit + Hysteresis Interval

1.1.1.5 AOOR Requirements

The AOOR mode paces the atrium at a variable interval dependent on the movement activity of the pacemaker user. The initial interval used is the Lower Rate Limit. The pulse is determined by the Atrial Pulse Width, defined by the user above. The more the user moves, the higher the pacing rate goes up to the Maximum Sensor Rate.

Given: The user sets the pacemaker to AOOR and inputs the required parameters

When: Current Pacing Rate has passed

Then: The pacemaker paces the atrium for an Atrial Pulse Width pulse

Given: The user sets the pacemaker to AOOR and inputs the required parameters

When: The user increases their Activity Level past the Activity Threshold

Then: The pacemaker increases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

Given: The user sets the pacemaker to AOOR and inputs the required parameters

When: The user their Activity Level past the Activity Threshold, then decreases their Activity Level

Then: The pacemaker decreases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

1.1.1.6 VOOR Requirements

The VOOR mode paces the ventricle at a variable interval dependent on the movement activity of the pacemaker user. The initial interval used is the Lower Rate Limit. The pulse is determined by the Ventricle Pulse Width, defined by the user above. The more the user moves, the higher the pacing rate goes up to the Maximum Sensor Rate.

Given: The user sets the pacemaker to VOOR and inputs the required parameters

When: Current Pacing Rate has passed

Then: The pacemaker paces the ventricle for a Ventricle Pulse Width pulse

Given: The user sets the pacemaker to VOOR and inputs the required parameters

When: The user increases their Activity Level past the Activity Threshold

Then: The pacemaker increases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

Given: The user sets the pacemaker to VOOR and inputs the required parameters

When: The user their Activity Level past the Activity Threshold, then decreases their Activity Level

Then: The pacemaker decreases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

1.1.1.7 AAIR Requirements

The AAIR mode paces the atrium at a variable interval dependent on the movement activity of the pacemaker user. The initial interval used is the Lower Rate Limit. The pulse is determined by the Atrial Pulse Width, defined by the user above. The more the user moves, the higher the pacing rate goes up to the Maximum Sensor Rate. If there is natural atrium heart activity between paces and after the Atrial Refraction Period, the AAIR mode does not pace for an interval defined by Current Pacing Rate + Hysteresis Interval. If there is natural atrium heart activity in this interval, the interval resets.

Given: The user sets the pacemaker to AAIR and inputs the required parameters

When: Current Pacing Rate has passed

Then: The pacemaker paces the atrium for an Atrial Pulse Width pulse

Given: The user sets the pacemaker to AAIR and inputs the required parameters

When: There has been natural atrium heart activity between paces and after the Atrial Refraction Period

Then: The pacemaker does not pace for an interval defined by Current Pacing Rate + Hysteresis Interval

Given: The user sets the pacemaker to AAIR and inputs the required parameters

When: The user increases their Activity Level past the Activity Threshold

Then: The pacemaker increases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

Given: The user sets the pacemaker to AAIR and inputs the required parameters

When: The user their Activity Level past the Activity Threshold, then decreases their Activity Level

Then: The pacemaker decreases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

1.1.1.8 VVIR Requirements

The VVIR mode paces the ventricle at a variable interval dependent on the movement activity of the pacemaker user. The initial interval used is the Lower Rate Limit. The pulse is determined by the Ventricle Pulse Width, defined by the user above. The more the user moves, the higher the pacing rate goes up to the Maximum Sensor Rate. If there is natural ventricle heart activity between paces and after the Ventricle Refraction Period, the VVIR mode does not pace for an interval defined by Current Pacing Rate + Hysteresis Interval. If there is natural ventricle heart activity in this interval, the interval resets.

Given: The user sets the pacemaker to VVIR and inputs the required parameters

When: Current Pacing Rate has passed

Then: The pacemaker paces the ventricle for a Ventricle Pulse Width pulse

Given: The user sets the pacemaker to VVIR and inputs the required parameters

When: There has been natural ventricle heart activity between paces and after the Ventricle Refraction Period

Then: The pacemaker does not pace for an interval defined by Current Pacing Rate + Hysteresis Interval

Given: The user sets the pacemaker to VVIR and inputs the required parameters

When: The user increases their Activity Level past the Activity Threshold

Then: The pacemaker increases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

Given: The user sets the pacemaker to VVIR and inputs the required parameters

When: The user their Activity Level past the Activity Threshold, then decreases their Activity Level

Then: The pacemaker decreases the current pacing rate over time to match the desired pacing rate based on the difference between the Activity Level and the Activity Threshold.

1.1.2 DCM

For the final requirements, the DCM is required to allow its user to interact with the pacemaker using a UART connection, so the pacing mode and parameters of the pacemaker can be transferred to the Pacemaker. These Parameters and pacing modes should be saved to the specific user so that on subsequent logins the user can reference and use their previously set parameters. The DCM also is required to receive Egram data from the pacemaker so it can display graphs for both the Atrial and Ventricular modes. The DCM is also required to securely store the user's username and password and have security so pacing parameters can be securely changed.

1.1.2.1 DCM Welcome Page

For the welcome page, the user must be able to register as a new user with a unique username and password. The DCM should also be able to store 10 users at once. If the application is closed, the credentials should stay stored. The DCM must also display the serial number of the Pacemaker connected and otherwise state that a device is not connected.

1.1.2.2 User Page

Once the user has logged in, The DCM must be able to display the pacing modes of the pacemaker. (AOO, AAI, AOOR, AAIR, VOO, VVI, VOOR and VVIR). The user page should also include a way to navigate to the Egram data screen and ways to navigate to the parameter modification screen for each pacing mode.

1.1.2.3 Parameter Page

A parameter page is required to allow the user to securely modify and then send their parameter data to the Pacemaker. This page requires a way to change all needed parameters for a given pacing mode. Each parameter should be incremented in a way specified in the Pacemaker documentation and be within its given value ranges. Once Parameters have been modified and saved, they must be sent over UART to the board to modify the Pacemakers operation.

1.1.2.4 Egram Page

The Egram page is required to graphically show the received Egram data for both Atrial and Ventricular modes from the Pacemaker. This page should have a scrolling graph showing real-time data being received via UART.

1.2 Design Decisions

1.2.1 Simulink

We used several subsystems within our system design to organize functions in a suitable manner. One of the subsystems is responsible for handling the mapping of input and output pins, which are responsible for transferring data used in the program, to their respective names. This mapping approach enhances the programs readability and hides the complexities of the underlying hardware. This approach allows the use of descriptive variable names like “ATR_CMP_DETECT instead of using generic labels like “D0” throughout the source code, making it clearer and separating the software from the hardware components. Hardware hiding also adds flexibility to the system by making it easier to use a different MCU or device for the same purpose. Other subsystems were used to implement functions such as UART communication with the DCM or calculating DPR. The subsystems approach not only makes our system more organized, but it also makes it more flexible and easier to adjust.

To make the system more efficient and reliable we use UART communication between the pacemaker and DCM to adjust the settings of the pacemaker. UART allows for bidirectional communication enabling the DCM to update pacemaker settings while receiving electrogram data to plot and give solid feedback to the users. The easy implementation of UART makes it a cost-effective solution for secure and real-time communication.

The pacemaker’s operational modes are implemented through a stateflow diagram, which selects the appropriate mode based on the input data. Additionally, nested stateflows are used to manage the different states needed for each mode. This approach enhances the system’s structure and clarity while ensuring that all modes are implemented and executed correctly.

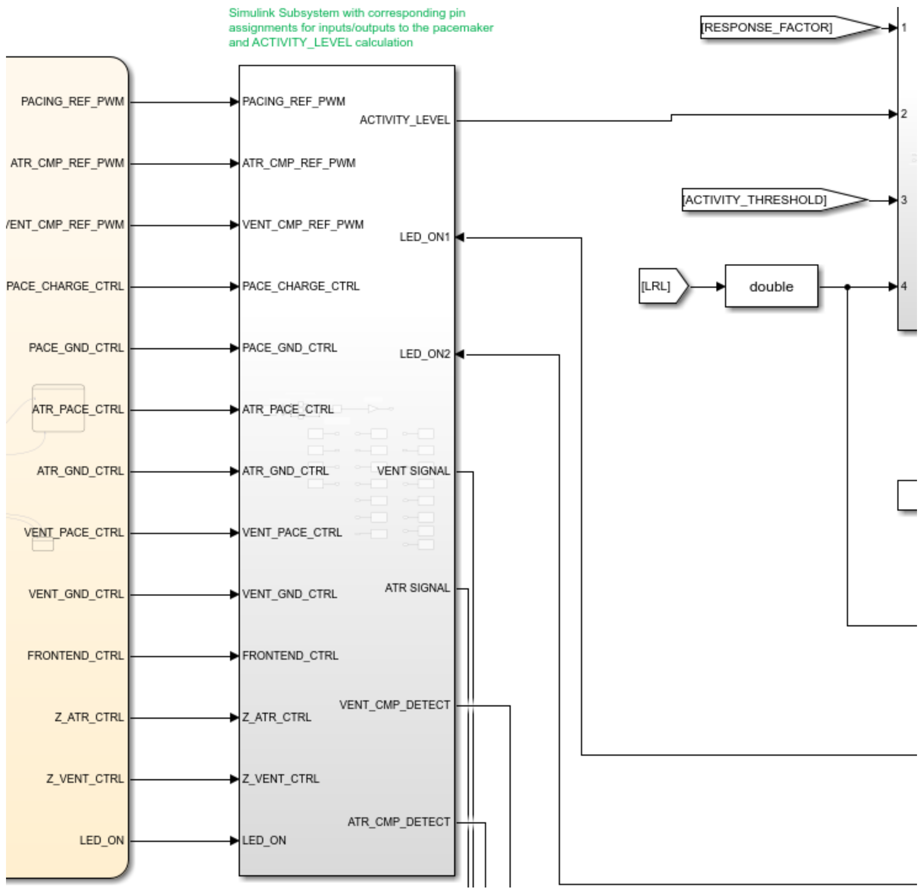


Figure 1: Connecting the pacemaker state flow to the subsystem

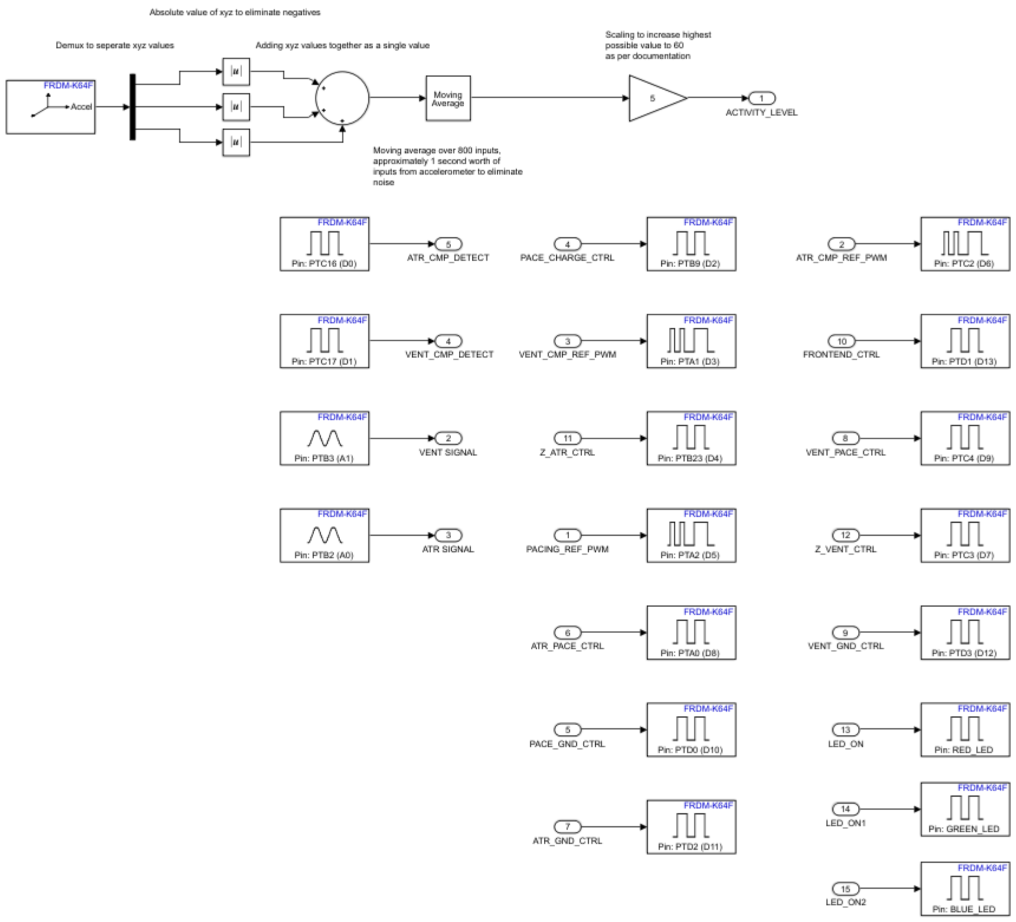


Figure 2: Pin assignments inside the subsystem along with accelerometer calculations

1.2.2 DCM

1.2.2.1 DCM Visuals

For the visual aspect of the DCM, we used Python’s Tkinter library to implement the GUI for the pacemaker. We chose this because of the intuitive nature of the library to be able to create the application promptly. Within the DCM itself, we had different screens, each placed within its own function. We have the welcome page which contains the login and registration aspect of the DCM. Once each button is pressed, a new screen function is called, and the current frame is cleared of all its widgets/contents and the new contents of the page are then placed onto the frame. A quit button is also available on both the user screen and the welcome screen as well as a back button between screens.

Inside the user screen, we added eight buttons to represent the states. The states each contained the user-specific parameter values. We chose to use sliders as the adjusting mechanism for a better user experience. These sliders were set within the appropriate ranges and increments for each parameter. We also added an egram button on the user screen which displays a matplotlib graph that contains the egram data that is received from the pacemaker. There are also new parameters added to specific states as per the project requirements for the rate adaptive mode states which the user can adjust and send to the pacemaker. We also added a “submit” button on each of the state screen. This was done to increase safety in the application, so if the user accidentally modified the parameters a second failsafe was in place to stop accidental data to be sent to the Pacemaker.

Issues that we faced while building the visuals and placing them on the frame were that the visuals were being put onto the frame as they were being instantiated. This caused getting data from the username and password boxes not possible. To fix this, we separated instantiation and placement on the frame so we are able to use the get() function in the Tkinter library. Issues that we faces was the implementation of the new buttons and fitting all new widgets on the same screen. The last time we were implementing visuals we were ensuring the visuals take a large amount of the space. We did not anticipate that there will be more parameters and buttons to add to the application. We also had issues integrating the matplotlib library with the tkinter library as the packing method of widgets did not match our old “grid” method of showing the widgets on the screen.

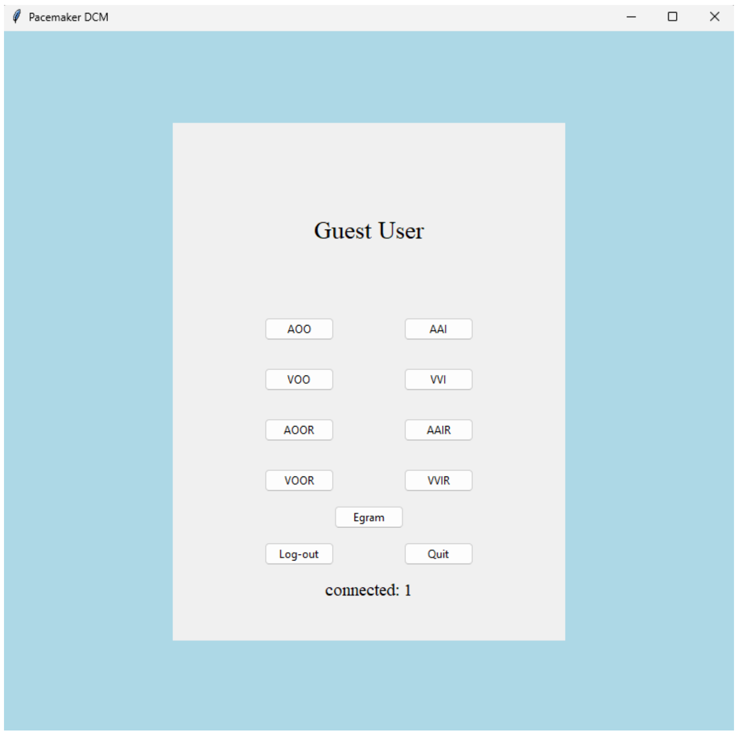
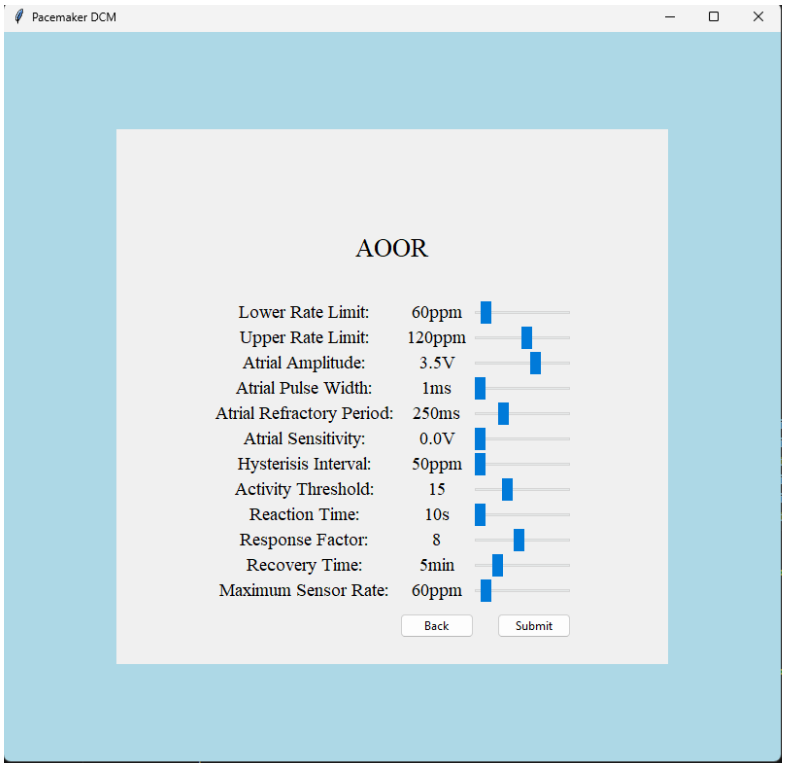


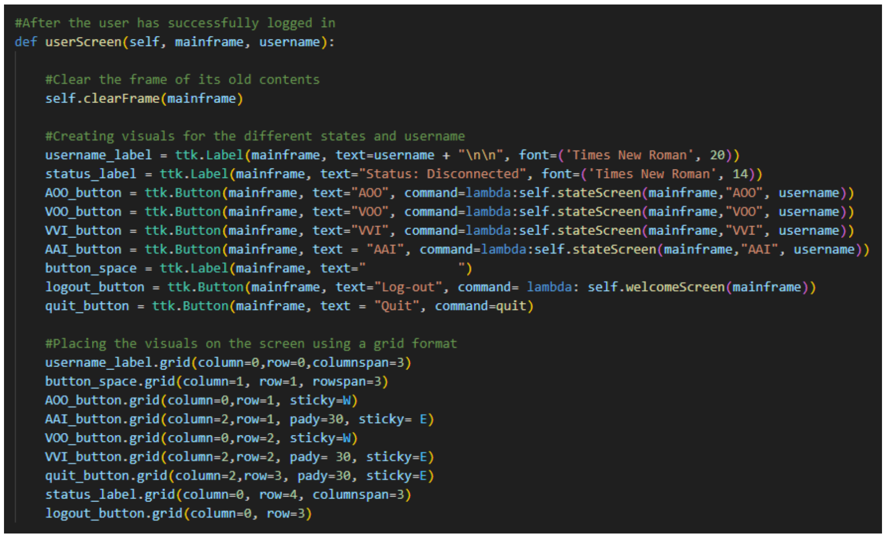
Figure 3: Home Page of DCM after Login



AOOR

Lower Rate Limit: 60ppm
Upper Rate Limit: 120ppm
Atrial Amplitude: 3.5V
Atrial Pulse Width: 1ms
Atrial Refractory Period: 250ms
Atrial Sensitivity: 0.0V
Hysterisis Interval: 50ppm
Activity Threshold: 15
Reaction Time: 10s
Response Factor: 8
Recovery Time: 5min
Maximum Sensor Rate: 60ppm

Figure 4: Sliders for adjusting Parameter Values



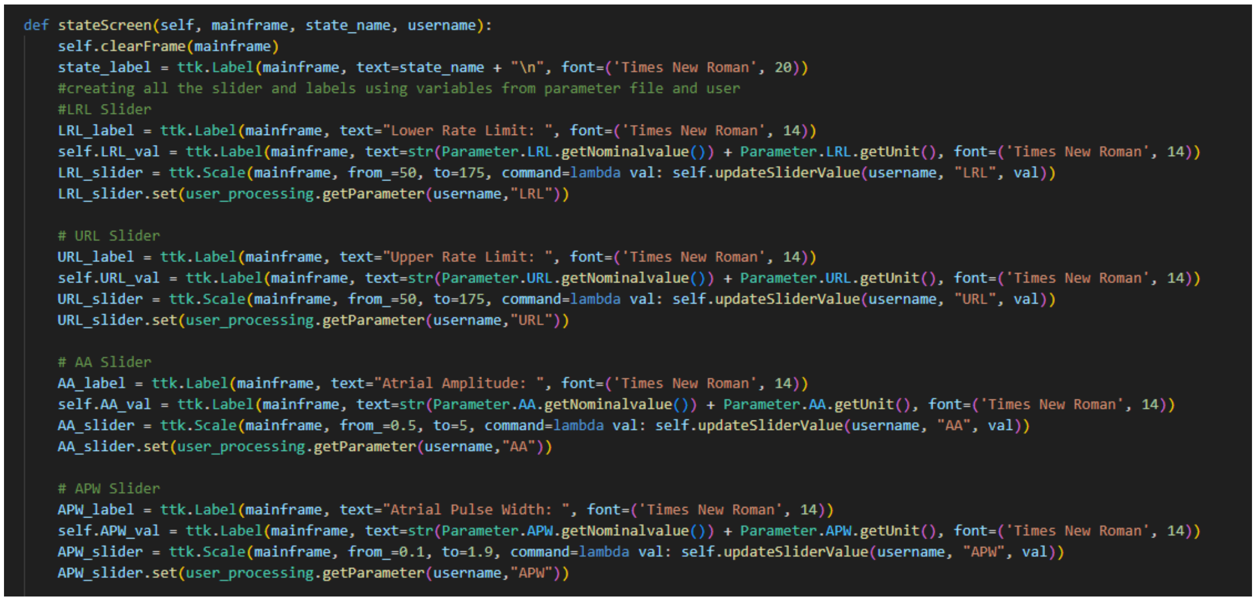
```
#After the user has successfully logged in
def userScreen(self, mainframe, username):

    #Clear the frame of its old contents
    self.clearFrame(mainframe)

    #Creating visuals for the different states and username
    username_label = ttk.Label(mainframe, text=username + "\n\n", font=('Times New Roman', 20))
    status_label = ttk.Label(mainframe, text="Status: Disconnected", font=('Times New Roman', 14))
    AOO_button = ttk.Button(mainframe, text="AOO", command=lambda:self.stateScreen(mainframe,"AOO", username))
    VOO_button = ttk.Button(mainframe, text="VOO", command=lambda:self.stateScreen(mainframe,"VOO", username))
    VVI_button = ttk.Button(mainframe, text="VVI", command=lambda:self.stateScreen(mainframe,"VVI", username))
    AAI_button = ttk.Button(mainframe, text = "AAI", command=lambda:self.stateScreen(mainframe,"AAI", username))
    button_space = ttk.Label(mainframe, text="           ")
    logout_button = ttk.Button(mainframe, text="Log-out", command= lambda: self.welcomeScreen(mainframe))
    quit_button = ttk.Button(mainframe, text = "Quit", command=quit)

    #Placing the visuals on the screen using a grid format
    username_label.grid(column=0,row=0,columnspan=3)
    button_space.grid(column=1, row=1, rowspan=3)
    AOO_button.grid(column=0,row=1, sticky=W)
    AAI_button.grid(column=2,row=1, pady=30, sticky= E)
    VOO_button.grid(column=0,row=2, sticky=W)
    VVI_button.grid(column=2,row=2, pady= 30, sticky=E)
    quit_button.grid(column=2,row=3, pady=30, sticky=E)
    status_label.grid(column=0, row=4, columnspan=3)
    logout_button.grid(column=0, row=3)
```

Figure 5: Separation of instantiation and placement of widgets



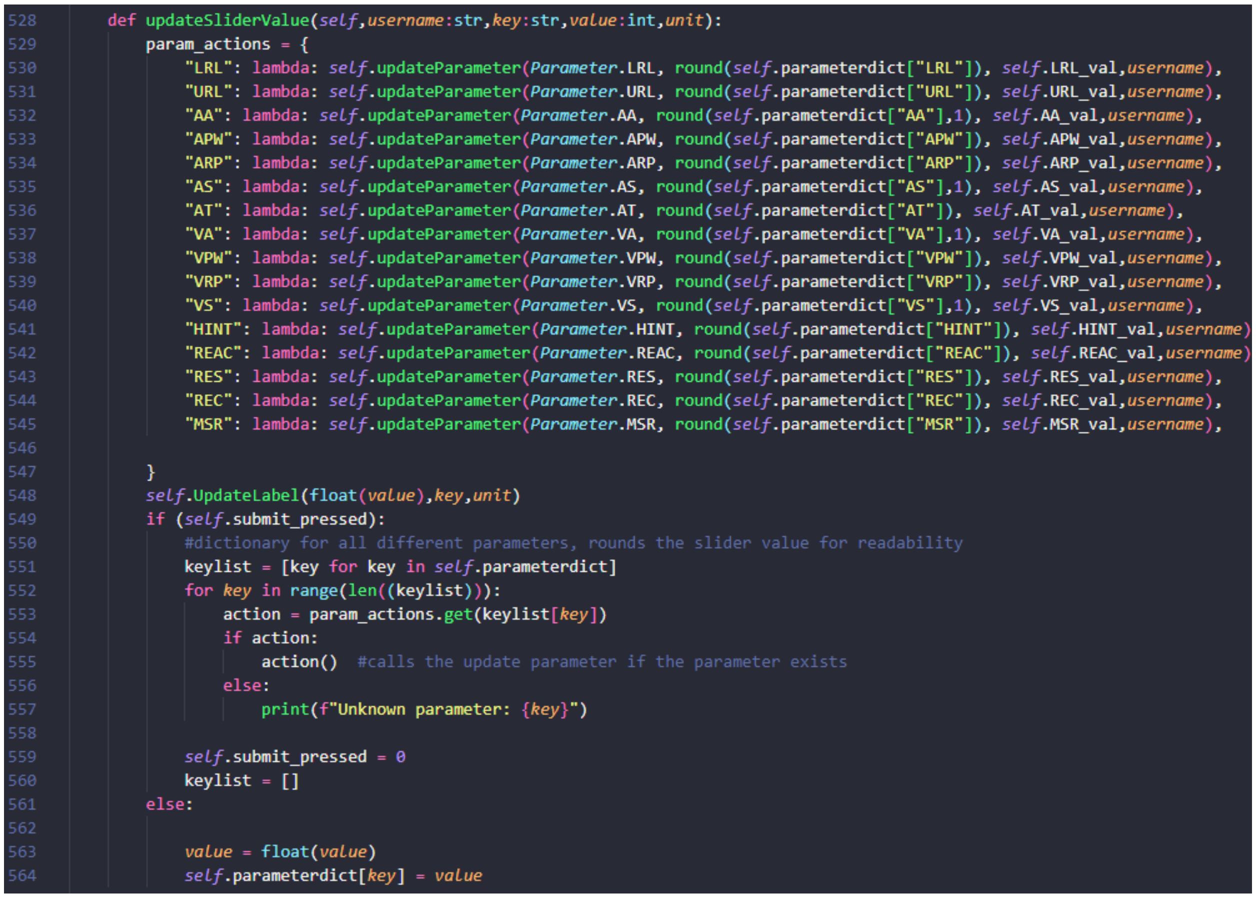
```python
def stateScreen(self, mainframe, state_name, username):
    self.clearFrame(mainframe)
    state_label = ttk.Label(mainframe, text=state_name + "\n", font=('Times New Roman', 20))
    #creating all the slider and labels using variables from parameter file and user
    #LRL Slider
    LRL_label = ttk.Label(mainframe, text="Lower Rate Limit: ", font=('Times New Roman', 14))
    self.LRL_val = ttk.Label(mainframe, text=str(Parameter.LRL.getNominalvalue()) + Parameter.LRL.getUnit(), font=('Times New Roman', 14))
    LRL_slider = ttk.Scale(mainframe, from_=50, to=175, command=lambda val: self.updateSliderValue(username, "LRL", val))
    LRL_slider.set(user_processing.getParameter(username,"LRL"))

    # URL Slider
    URL_label = ttk.Label(mainframe, text="Upper Rate Limit: ", font=('Times New Roman', 14))
    self.URL_val = ttk.Label(mainframe, text=str(Parameter.URL.getNominalvalue()) + Parameter.URL.getUnit(), font=('Times New Roman', 14))
    URL_slider = ttk.Scale(mainframe, from_=50, to=175, command=lambda val: self.updateSliderValue(username, "URL", val))
    URL_slider.set(user_processing.getParameter(username,"URL"))

    # AA Slider
    AA_label = ttk.Label(mainframe, text="Atrial Amplitude: ", font=('Times New Roman', 14))
    self.AA_val = ttk.Label(mainframe, text=str(Parameter.AA.getNominalvalue()) + Parameter.AA.getUnit(), font=('Times New Roman', 14))
    AA_slider = ttk.Scale(mainframe, from_=0.5, to=5, command=lambda val: self.updateSliderValue(username, "AA", val))
    AA_slider.set(user_processing.getParameter(username,"AA"))

    # APW Slider
    APW_label = ttk.Label(mainframe, text="Atrial Pulse Width: ", font=('Times New Roman', 14))
    self.APW_val = ttk.Label(mainframe, text=str(Parameter.APW.getNominalvalue()) + Parameter.APW.getUnit(), font=('Times New Roman', 14))
    APW_slider = ttk.Scale(mainframe, from_=0.1, to=1.9, command=lambda val: self.updateSliderValue(username, "APW", val))
    APW_slider.set(user_processing.getParameter(username,"APW"))
```

Figure 6: Setting up the sliders in the code



```python
    def updateSliderValue(self,username:str,key:str,value:int,unit):
        param_actions = {
            "LRL": lambda: self.updateParameter(Parameter.LRL, round(self.parameterdict["LRL"]), self.LRL_val,username),
            "URL": lambda: self.updateParameter(Parameter.URL, round(self.parameterdict["URL"]), self.URL_val,username),
            "AA": lambda: self.updateParameter(Parameter.AA, round(self.parameterdict["AA"],1), self.AA_val,username),
            "APW": lambda: self.updateParameter(Parameter.APW, round(self.parameterdict["APW"]), self.APW_val,username),
            "ARP": lambda: self.updateParameter(Parameter.ARP, round(self.parameterdict["ARP"]), self.ARP_val,username),
            "AS": lambda: self.updateParameter(Parameter.AS, round(self.parameterdict["AS"],1), self.AS_val,username),
            "AT": lambda: self.updateParameter(Parameter.AT, round(self.parameterdict["AT"]), self.AT_val,username),
            "VA": lambda: self.updateParameter(Parameter.VA, round(self.parameterdict["VA"],1), self.VA_val,username),
            "VPW": lambda: self.updateParameter(Parameter.VPW, round(self.parameterdict["VPW"]), self.VPW_val,username),
            "VRP": lambda: self.updateParameter(Parameter.VRP, round(self.parameterdict["VRP"]), self.VRP_val,username),
            "VS": lambda: self.updateParameter(Parameter.VS, round(self.parameterdict["VS"],1), self.VS_val,username),
            "HINT": lambda: self.updateParameter(Parameter.HINT, round(self.parameterdict["HINT"]), self.HINT_val,username)
            "REAC": lambda: self.updateParameter(Parameter.REAC, round(self.parameterdict["REAC"]), self.REAC_val,username)
            "RES": lambda: self.updateParameter(Parameter.RES, round(self.parameterdict["RES"]), self.RES_val,username),
            "REC": lambda: self.updateParameter(Parameter.REC, round(self.parameterdict["REC"]), self.REC_val,username),
            "MSR": lambda: self.updateParameter(Parameter.MSR, round(self.parameterdict["MSR"]), self.MSR_val,username),

        }
        self.UpdateLabel(float(value),key,unit)
        if (self.submit_pressed):
            #dictionary for all different parameters, rounds the slider value for readability
            keylist = [key for key in self.parameterdict]
            for key in range(len((keylist))):
                action = param_actions.get(keylist[key])
                if action:
                    action()  #calls the update parameter if the parameter exists
                else:
                    print(f"Unknown parameter: {key}")

            self.submit_pressed = 0
            keylist = []
        else:

            value = float(value)
            self.parameterdict[key] = value
```

Figure 7: Functions to update slider values and json values

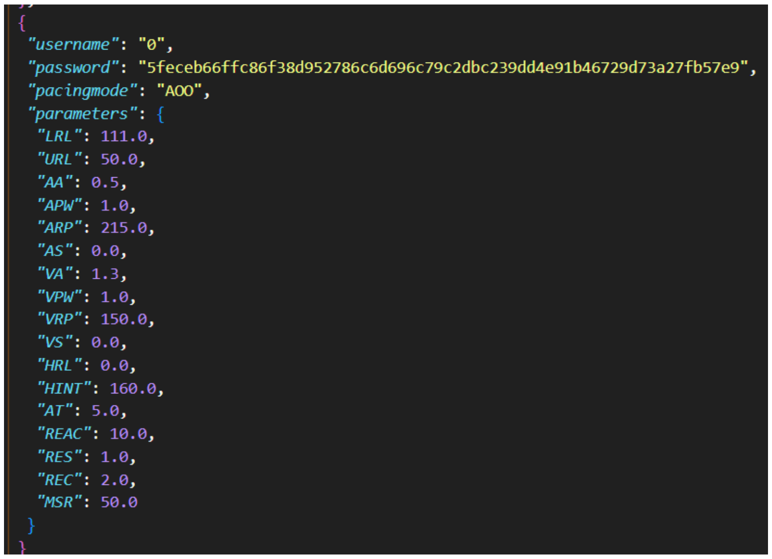
1.2.2.2 DCM User Storage and Parameters

For the backend and user information portion of the DCM, we used the JSON library to hold user data. JSON files support a quick and easy way to write and read data in formats that Python can interpret. We decided to store our users' usernames, passwords, pacing mode and parameters in the JSON file using a dictionary type that is stored inside a list. This way we could easily store data values while also storing the name of the value to reference what the data was. This also lets us reference each user as a list index, making it simple to check if the user login is correct and change user parameters. In Figure 8 the structure of the JSON file information is shown. We also chose to store the parameters as a dictionary as well so each of the different parameters can be accessed separately. In Figure 9 the parameter python file is shown, each parameter is instantiated at the start of the code so each separate parameter can have a name, value and unit.

The user storage also implements Sha256 hashing using the Hashlib library. With this hashing the passwords become more securely stored in the JSON file as they are hashed prior to being written and cannot be seen as plain text in the file. A hashed password can be seen in Figure 8, greatly increasing the security of the password storage.

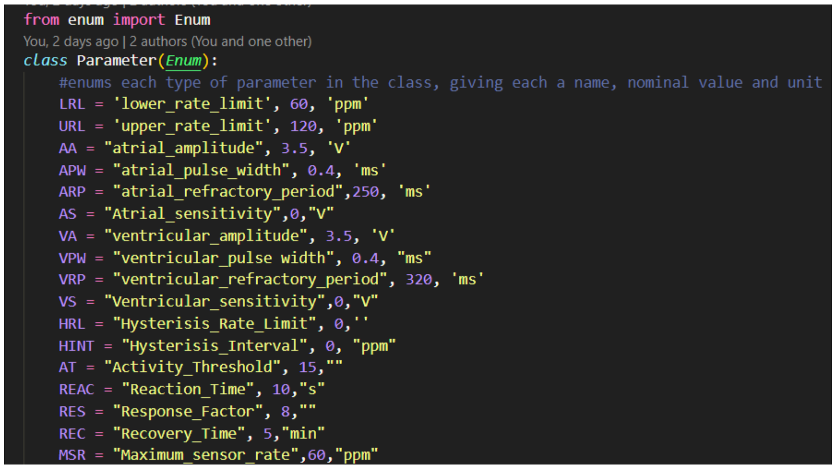
We also chose to code our DCM using mainly object-oriented programming and multiple Python files, the user, storage system, processing system, parameters, UART and interface all have their own Python file and are made up of classes and functions. This makes our code extremely modular and easy to understand.

A major issue we faced was users were able to submit the exact same username as an existing user. This did not work with our current password-checking function as it would check the inputted password to the first instance of the username in the list and never the second one, meaning the second user would never be able to log in. To fix this we decided that all usernames needed to be unique and added messaging when a registered user inputted a username that was the same as an existing one. To check if the user login is correct (Figure 10), we first implemented a function to check if the username existed in the JSON file (Figure 11). If this function returns true, then a separate function is called to check what index in the list that user was (Figure 12). This index is then used in a password-checking function (Figure 13) to see if the password at that index in the JSON file matches the password inputted by the user. The register check is similar to the login check but instead, it checks that the username does not already exist and if the number of users is below 10, this is shown in Figure 14.



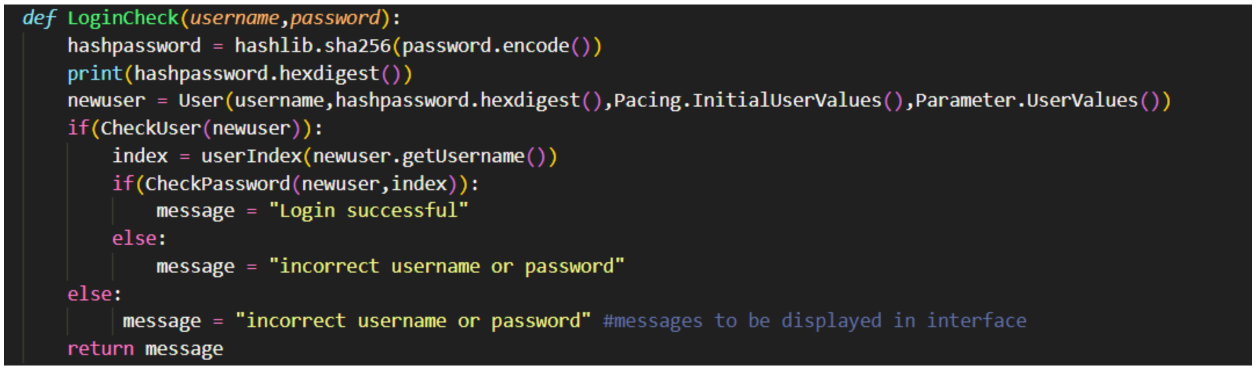
```
{
  "username": "0",
  "password": "5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9",
  "pacingmode": "AOO",
  "parameters": {
   "LRL": 111.0,
   "URL": 50.0,
   "AA": 0.5,
   "APW": 1.0,
   "ARP": 215.0,
   "AS": 0.0,
   "VA": 1.3,
   "VPW": 1.0,
   "VRP": 150.0,
   "VS": 0.0,
   "HRL": 0.0,
   "HINT": 160.0,
   "AT": 5.0,
   "REAC": 10.0,
   "RES": 1.0,
   "REC": 2.0,
   "MSR": 50.0
  }
}
```

Figure 8. Json file format



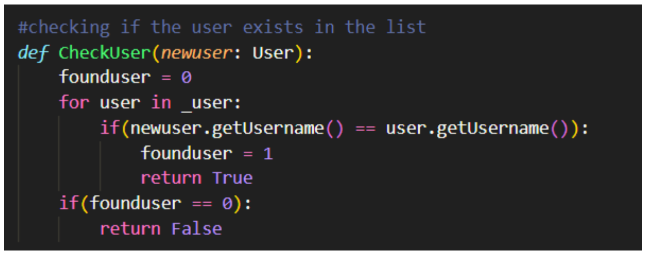
```
from enum import Enum
You, 2 days ago | 2 authors (You and one other)
class Parameter(Enum):
    #enums each type of parameter in the class, giving each a name, nominal value and unit
    LRL = 'lower_rate_limit', 60, 'ppm'
    URL = 'upper_rate_limit', 120, 'ppm'
    AA = "atrial_amplitude", 3.5, 'V'
    APW = "atrial_pulse_width", 0.4, 'ms'
    ARP = "atrial_refractory_period",250, 'ms'
    AS = "Atrial_sensitivity",0,"V"
    VA = "ventricular_amplitude", 3.5, 'V'
    VPW = "ventricular_pulse width", 0.4, "ms"
    VRP = "ventricular_refractory_period", 320, 'ms'
    VS = "Ventricular_sensitivity",0,"V"
    HRL = "Hysterisis_Rate_Limit", 0,''
    HINT = "Hysterisis_Interval", 0, "ppm"
    AT = "Activity_Threshold", 15,""
    REAC = "Reaction_Time", 10,"s"
    RES = "Response_Factor", 8,""
    REC = "Recovery_Time", 5,"min"
    MSR = "Maximum_sensor_rate",60,"ppm"
```

Figure 9. Parameter file



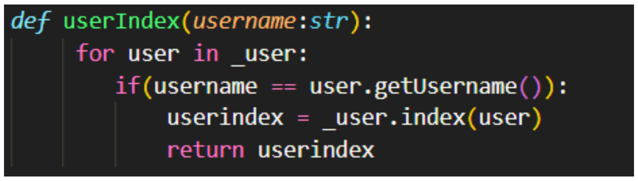
```
def LoginCheck(username,password):
    hashpassword = hashlib.sha256(password.encode())
    print(hashpassword.hexdigest())
    newuser = User(username,hashpassword.hexdigest(),Pacing.InitialUserValues(),Parameter.UserValues())
    if(CheckUser(newuser)):
        index = userIndex(newuser.getUsername())
        if(CheckPassword(newuser,index)):
            message = "Login successful"
        else:
            message = "incorrect username or password"
    else:
        message = "incorrect username or password" #messages to be displayed in interface
    return message
```

Figure 10. User Login Check



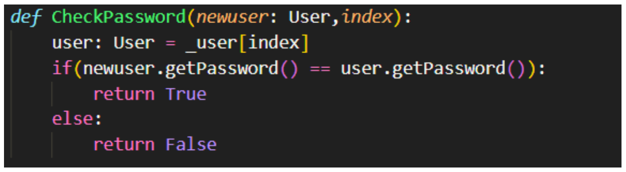
```python
#checking if the user exists in the list
def CheckUser(newuser: User):
    founduser = 0
    for user in _user:
        if(newuser.getUsername() == user.getUsername()):
            founduser = 1
            return True
    if(founduser == 0):
        return False
```

Figure 11. Checking if User Exists



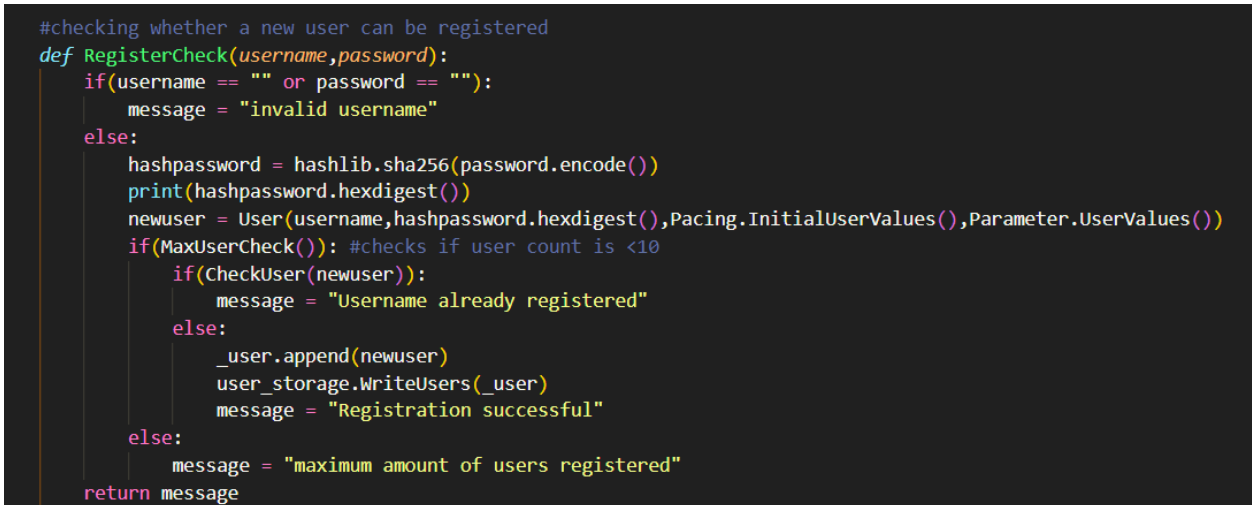
```python
def userIndex(username:str):
    for user in _user:
        if(username == user.getUsername()):
            userindex = _user.index(user)
            return userindex
```

Figure 12. Finding Index of User



```python
def CheckPassword(newuser: User,index):
    user: User = _user[index]
    if(newuser.getPassword() == user.getPassword()):
        return True
    else:
        return False
```

Figure 13. Password Check Function



```python
#checking whether a new user can be registered
def RegisterCheck(username,password):
    if(username == "" or password == ""):
        message = "invalid username"
    else:
        hashpassword = hashlib.sha256(password.encode())
        print(hashpassword.hexdigest())
        newuser = User(username,hashpassword.hexdigest(),Pacing.InitialUserValues(),Parameter.UserValues())
        if(MaxUserCheck()): #checks if user count is <10
            if(CheckUser(newuser)):
                message = "Username already registered"
            else:
                _user.append(newuser)
                user_storage.WriteUsers(_user)
                message = "Registration successful"
        else:
            message = "maximum amount of users registered"
    return message
```

Figure 14. Registration Check Function

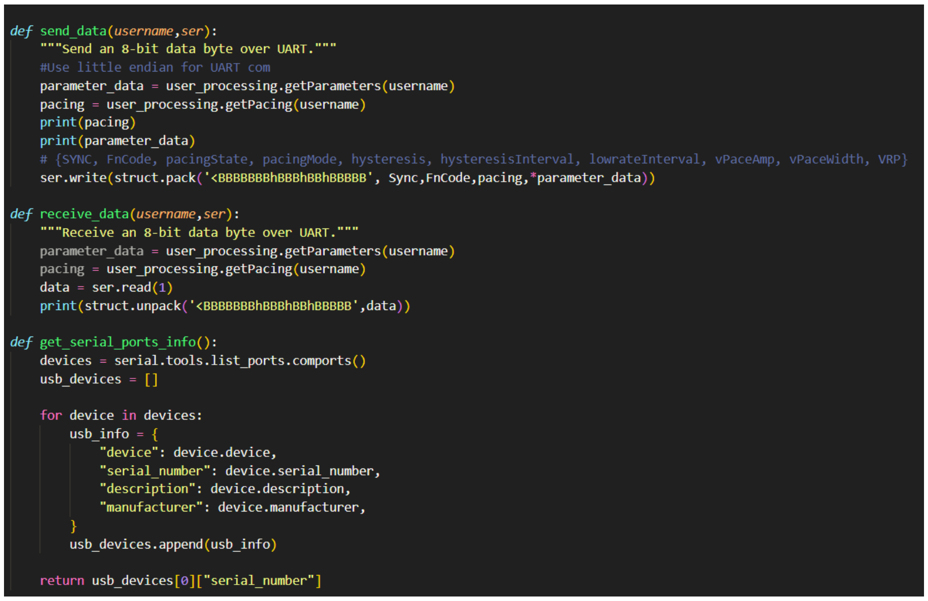
1.2.2.3 UART Communication

For communicating with the board, we were required to use UART to send and receive data between the DCM and the FRDM board. For this we decided to use the Pyserial, Struct and the serial tools library. The Pyserial library gives us an easy way to connect to the COM ports on the computer to send data over UART. This lets us specify which port we are sending data on and when we want to read and write from and to the Pacemaker.

The Struct library was used to interpret our integers and floats for the parameter data and convert it into byte data types so that it could be sent over UART. When using the Struct.pack() function we could convert our data into byte strings of hex to then be sent over UART. For receiving we used a similar method of using the struct.unpack() function where we would take a hex byte string from the Pacemaker and unpack it back into its original decimal integer or float values.

For checking the connected device, we used the serial.tools library. This library can directly access the information on what port a device is connected to and other important information like its serial id and name. With this we could check which port the Pacemaker device was connected to and then access its serial ID to display in the DCM. We could also use this to check if no device was connected and then display to the user that their Pacemaker device is not plugged in or plugged in improperly.

An issue we faced with UART communication was that the Pacemaker would not send data back fast enough. The DCM would send UART data and then receive the same data back to verify the connection but many times the Pacemaker would send back empty byte strings to the DCM. To fix this we tried to add delays to the reading from the Pacemaker, but ultimately our fix was to only start reading data when a certain start value was eventually sent from the Pacemaker. Another issue we had was the sizing of the variables. Some parameters like the Atrial and Ventricular Amplitude were float data types and had decimals in them, while many of the other parameters were only integers. This made it difficult to send them all at once as some values needed to be scaled and some were sent as a normal value. To fix this we scaled all our decimal parameters by 10x so that they would no longer contain any decimals. Then when the values were received by the Pacemaker the values could be scaled back down to their original values.



```python
def send_data(username,ser):
    """Send an 8-bit data byte over UART."""
    #Use little endian for UART com
    parameter_data = user_processing.getParameters(username)
    pacing = user_processing.getPacing(username)
    print(pacing)
    print(parameter_data)
    # {SYNC, FnCode, pacingState, pacingMode, hysteresis, hysteresisInterval, lowrateInterval, vPaceAmp, vPaceWidth, VRP}
    ser.write(struct.pack('<BBBBBBBhBBBhBBhBBBBB', Sync,FnCode,pacing,*parameter_data))

def receive_data(username,ser):
    """Receive an 8-bit data byte over UART."""
    parameter_data = user_processing.getParameters(username)
    pacing = user_processing.getPacing(username)
    data = ser.read(1)
    print(struct.unpack('<BBBBBBBhBBBhBBhBBBBB',data))

def get_serial_ports_info():
    devices = serial.tools.list_ports.comports()
    usb_devices = []

    for device in devices:
        usb_info = {
            "device": device.device,
            "serial_number": device.serial_number,
            "description": device.description,
            "manufacturer": device.manufacturer,
        }
        usb_devices.append(usb_info)

    return usb_devices[0]["serial_number"]
```

Figure 15: UART sending and receiving, Serial port information

1.3 Validation

1.3.1 Simulink

For Simulink, as the current system passes all test cases and functions appropriately based on the provided documents, and it follows all requirements (as seen below), the requirements are valid. Requirements have remained the same throughout Assignments 1 and 2, and thus have remained correct.

1.3.2 DCM

The DCM system has met all the previously stated requirements, having multiple screens, a working login and registration page, status bar for UART connection, sending parameters over UART, a way to store and modify parameters, graphed Egram data and a way to save user values securely. As seen below in testing and verification, all tests for usernames, passwords, parameters, JSON file storage and hashing has passed. Egram did not pass our validation testing as the UART data for the graph was not interpreted properly.

1.4 Testing and Verification

1.4.1 Simulink Testing and Verification

To test the Simulink state flow, each mode was compiled onto the pacemaker separately with the corresponding test case parameters. Heartview was used to verify the operation of the pacemaker mode against the expected result. The pacemaker's built-in LED was also used as a secondary form of informal testing to ensure that the pacemaker was pacing accordingly.

The input parameters to the pacemaker for testing each mode is listed as follows:

Beats per minute(bpm) = 60 (+120 for adaptive rate),
Atrium pulse width = 1ms,
Ventricle pulse width = 1ms,
Atrial_Amplitude = 3.5V,
Ventricle_Amplitude = 3.5V,
ARP = 250ms,
VRP = 320ms,
Lower_Rate_Limit = 1000ms,
Upper_Rate_Limit = 500ms,
Hysteresis = 300ms,
ATR_CMP_REF_PWM = 90%,
VENT_CMP_REF_PWM = 90%,
Natural AV Delay = 30ms

For AOO and VOO testing, it is expected that the pacemaker can replicate the timing of the natural Atrial/Ventricle as there is no sensing in either of the two modes.

The natural atrium/ventricle was enabled and set to the same parameters listed previously. As seen in figure 15 for AOO or figure 16 for VOO, the natural signal (Red) maintains equal distance from the pacemaker signal (Blue). This confirms the expected behavior since the bpm and pulse width are identical for both the natural pulse and for the pacemaker.

This also verifies the requirement of the pacemaker pulsing after the lower rate limit for both AOO and VOO.

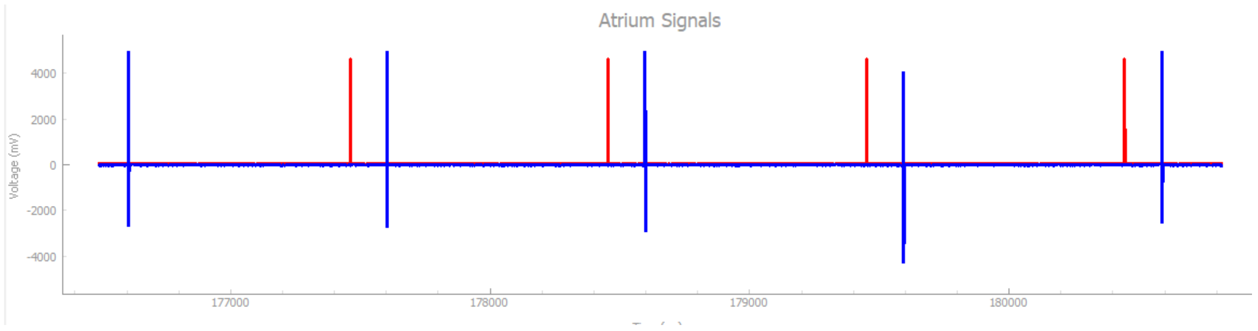


Figure 16: AOO mode with Natural Atrium Enabled

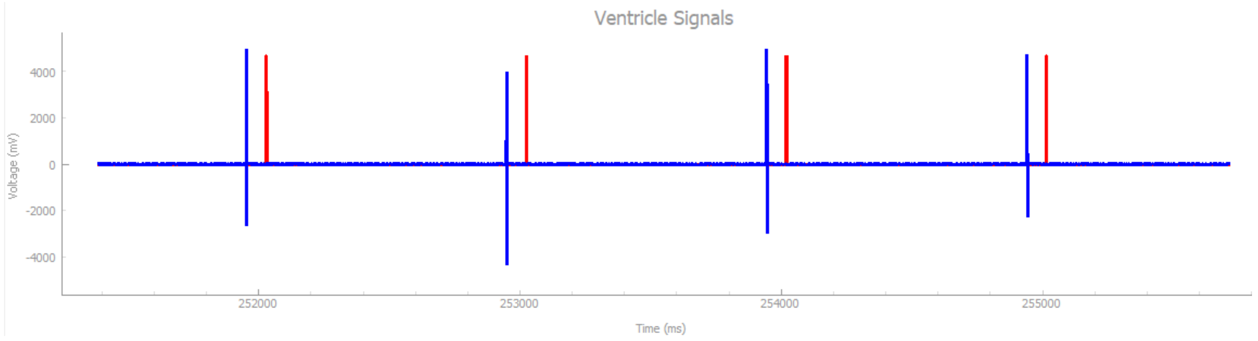


Figure 17: VOO mode with Natural Ventricle Enabled

For AAI and VVI testing, it is expected that the pacemaker will inhibit pacing if it senses a natural heartbeat. Both modes should also apply an additional hysteresis delay after sensing a natural heartbeat before attempting to pace to encourage self-pacing.

The natural atrium was enabled and set to the same parameters listed previously. As seen in figure 17 for AAI, the pacemaker inhibits pacing once a natural heartbeat (Red) is sensed confirming the first expected behavior. In figure 18 for AAI, the time between a natural heartbeat and a pace (Blue) is delayed by an additional hysteresis delay confirming the second expected behavior.

This test also verifies the requirements for AAI as the pacemaker paces at a rate of lower rate limit only when no natural pulse is detected. In addition, when a natural pulse is detected, the pacemaker waits an additional hysteresis delay before attempting to pace to allow for another natural pulse to occur.

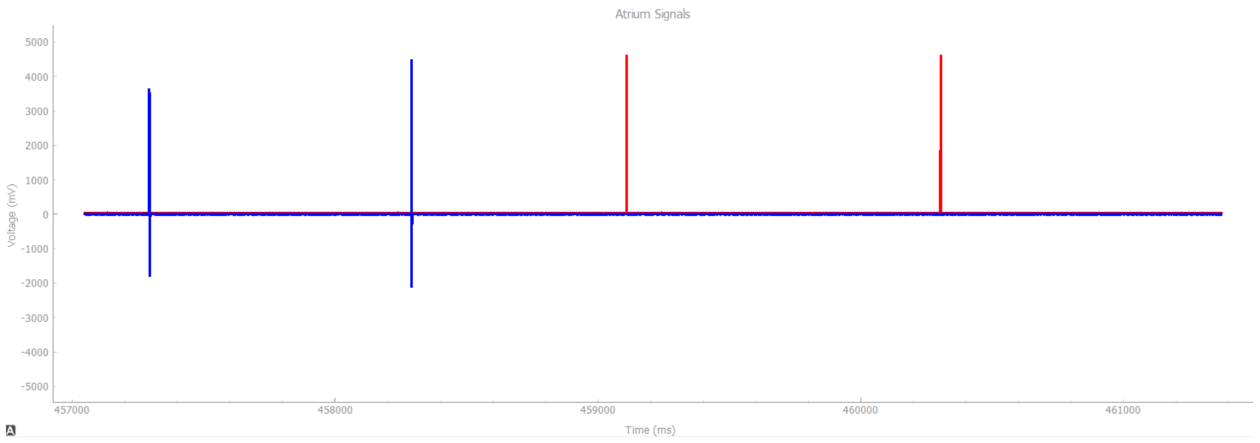


Figure 18: AAI mode with Natural Atrial Enabled during pacemaker pacing

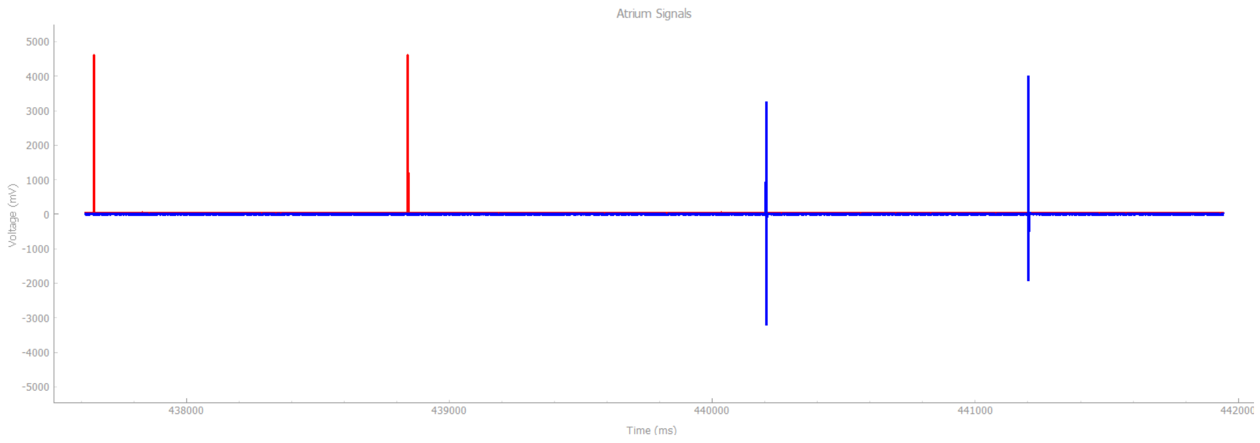


Figure 19: AAI mode with Natural Atrium Disabled to resume pacemaker pacing

The natural ventricle was enabled and set to the same parameters listed previously. As seen in figure 19 for VVI, the pacemaker inhibits pacing once a natural heartbeat (Red) is sensed confirming the first expected behavior. In figure 20 for VVI, the time between a natural heartbeat and a pace (Blue) is delayed by an additional hysteresis delay confirming the second expected behavior.

This test also verifies the requirements for VVI as the pacemaker paces at a rate of lower rate limit only when no natural pulse is detected. In addition, when a natural pulse is detected, the pacemaker waits an additional hysteresis delay before attempting to pace to allow for another natural pulse to occur.

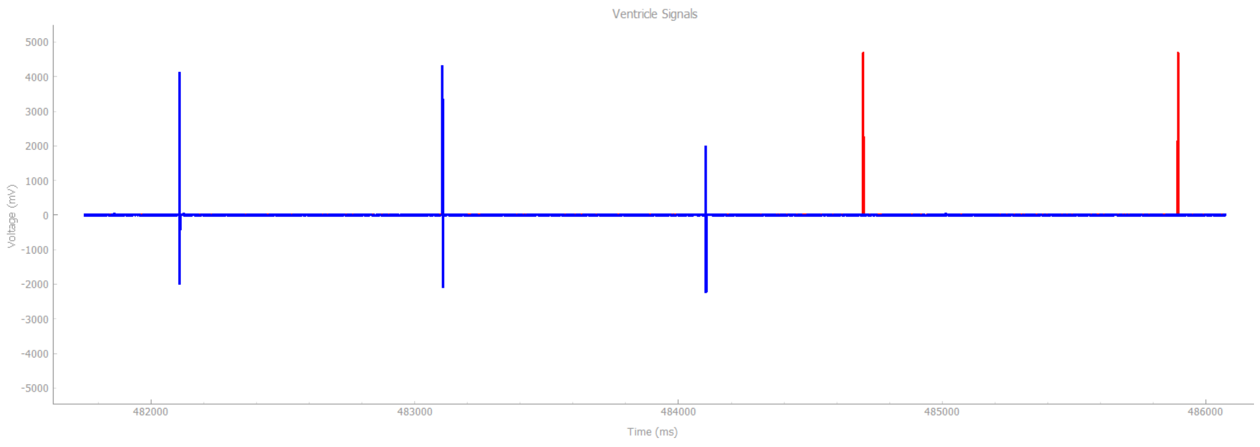


Figure 20: VVI mode with Natural Ventricle Enabled during pacemaker pacing

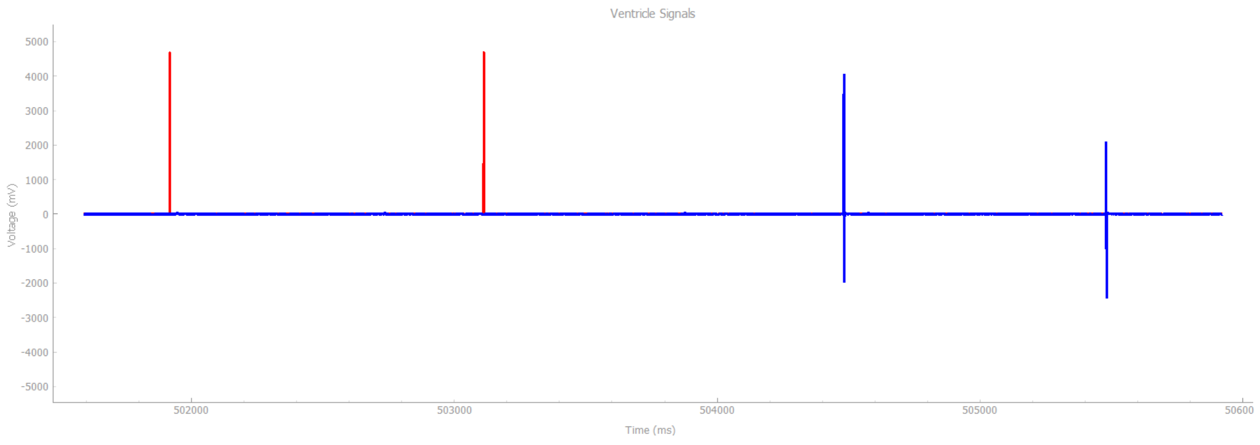


Figure 21: VVI mode with Natural Ventricle Disabled to resume pacemaker pacing

Rate Adaptive Modes

The rate adaptive modes were tested to ensure that after implementation, the modes behaved as they did before in a resting state. To do this, we kept the activity level below the activity threshold and ran the below tests. It is expected that each mode should behave exactly as its non rate adaptive counter part. The parameters stated at the beginning of the section are used to compare against the pacemaker output.

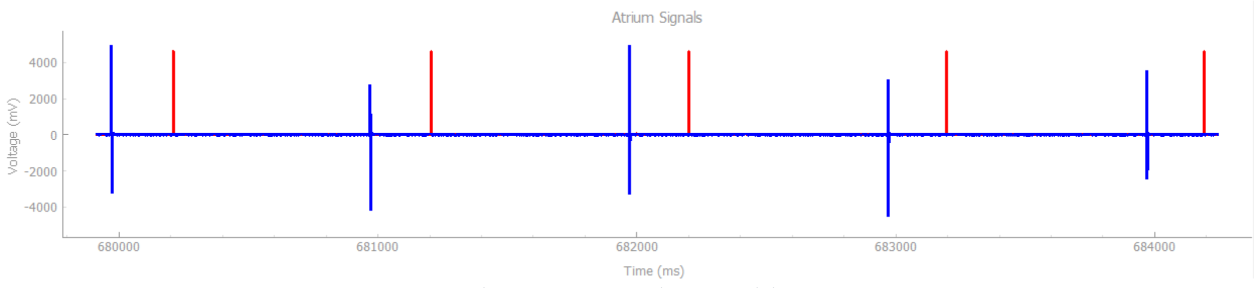


Figure 22: AOOR with no activity

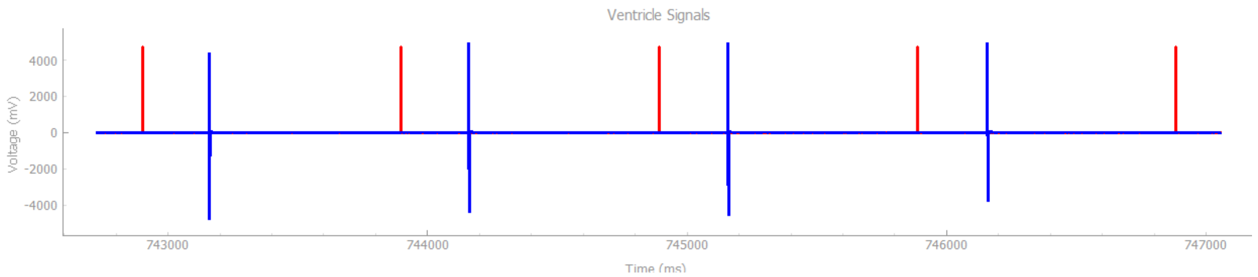


Figure 23: VOOR with no activity

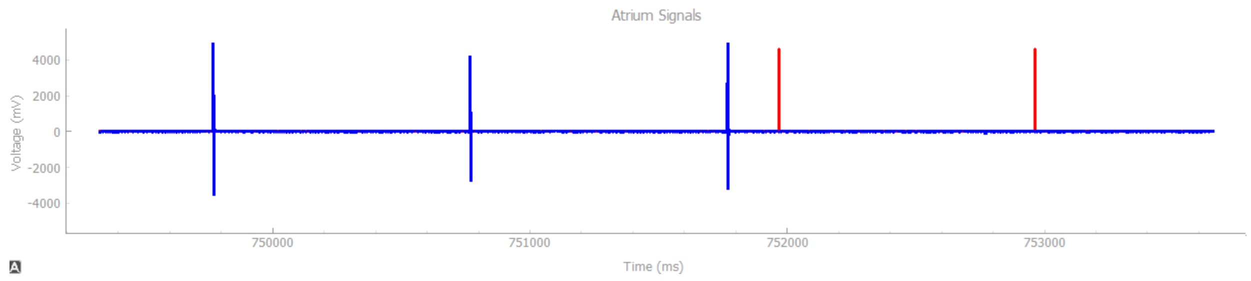


Figure 24: AAIR with no activity, paces until natural pace sensed

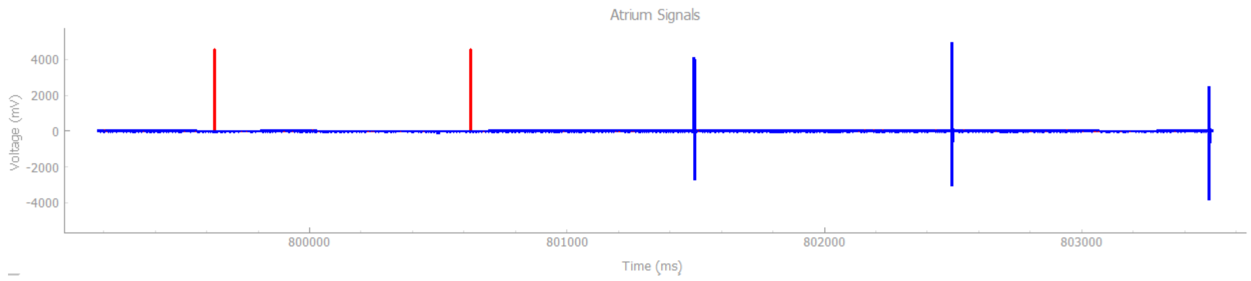


Figure 25: AAIR with no activity, starts pacing when no natural pace sensed

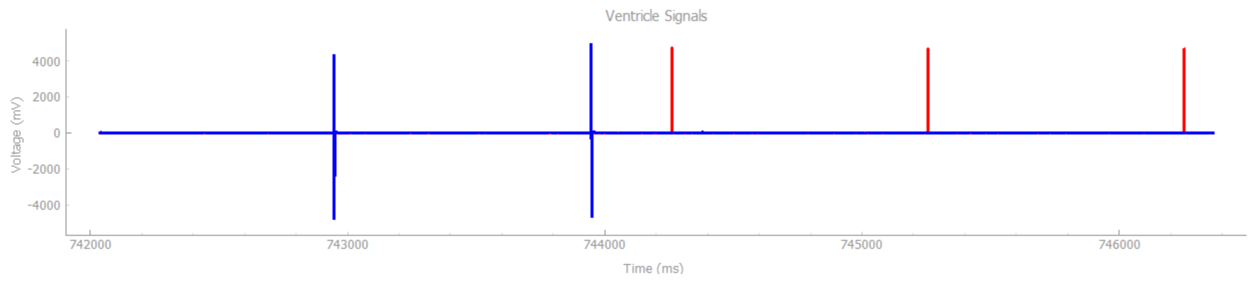


Figure 26: VVIR with no activity, stops pacing when natural pace sensed

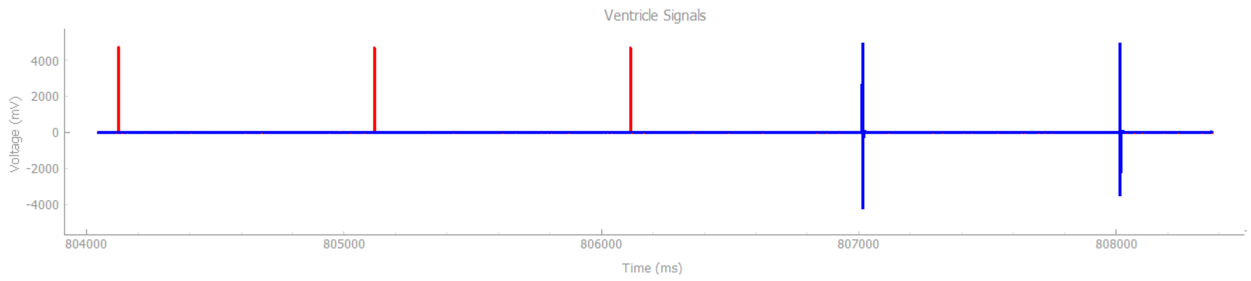


Figure 27: VVIR with no activity, paces when no natural heartbeat is sensed

This test confirms that all rate adaptive modes are functioning as expected during resting since no change in bpm is observed. This test also confirms the requirement for all rate adaptive modes that if no activity is observed, pace at the lower rate limit.

Next, testing was done to ensure that the pacing rate changes over time if the activity level is above the activity threshold. To do this, we ran the tests by shaking the Pacemaker vigorously for a set reaction time (10 seconds). It is expected that each rate adaptive mode should have its pacing increased to 120 bpm during the activity. The parameters used to verify this test are the same as the beginning of the section except bpm is increased to 120 bpm.

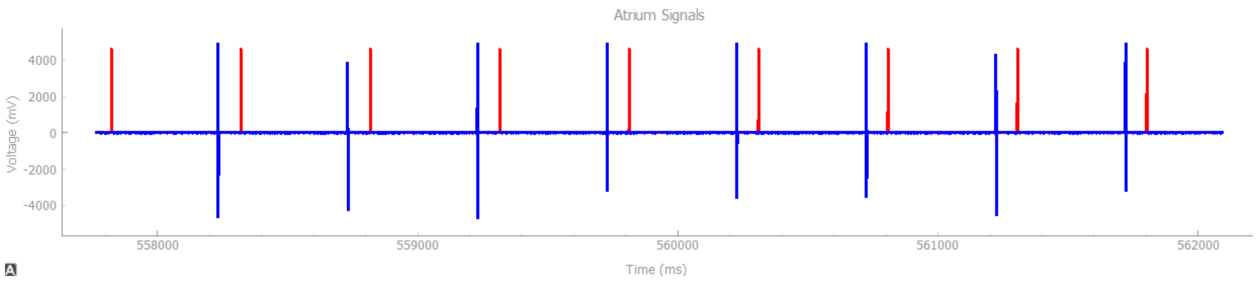


Figure 28: AOOR with activity

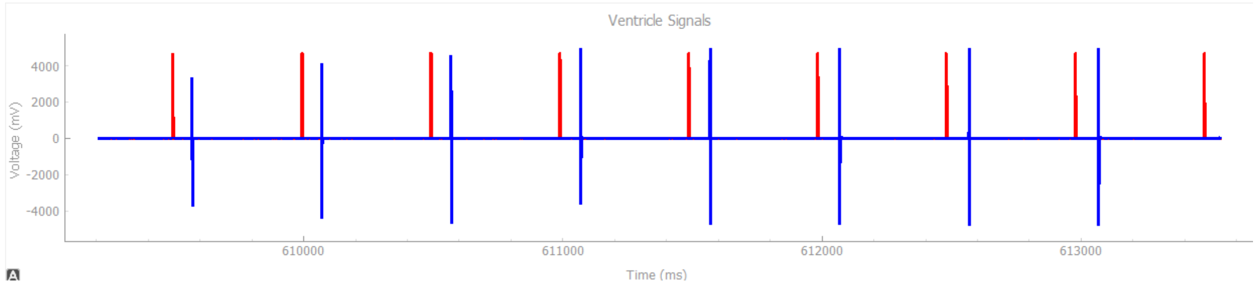


Figure 29: VOOR with activity

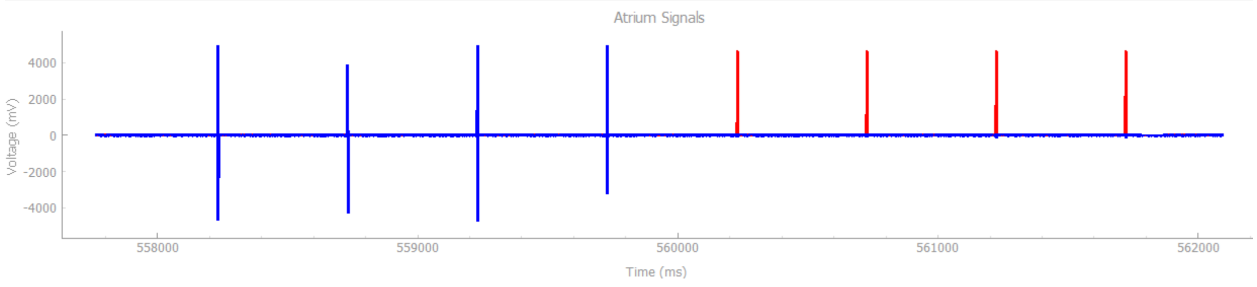


Figure 30: AAIR with activity, paces until natural pace is sensed

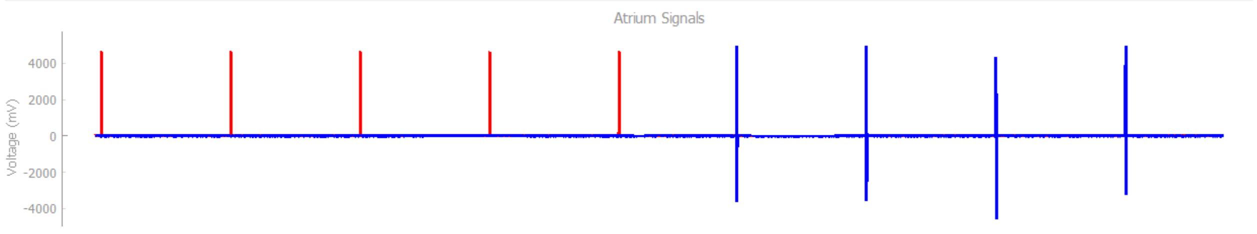


Figure 31: AAIR with activity, does not pace until natural pacing stops

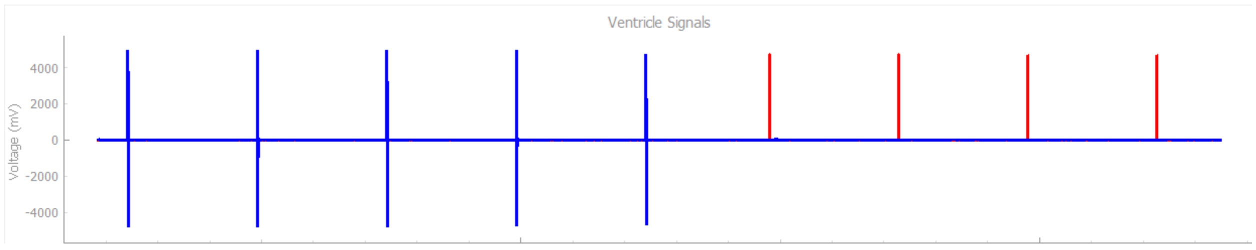


Figure 32: VVIR with activity, paces until natural pace sensed

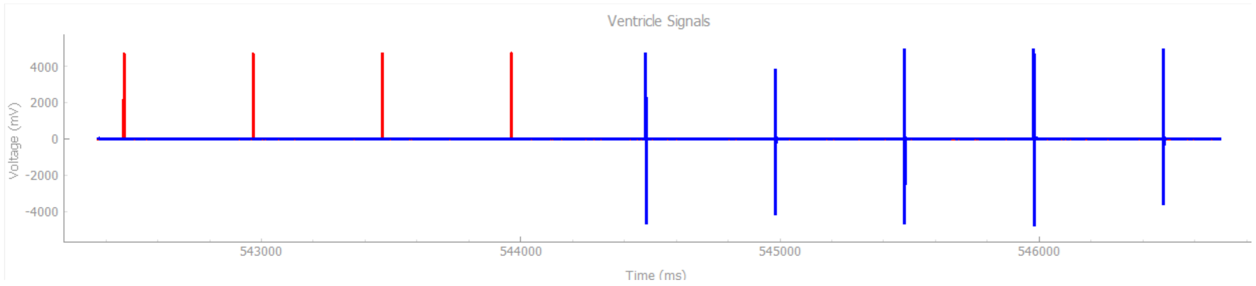


Figure 33: VVIR with activity, does not pace until natural pacing stops

It is observed that each rate adaptive mode increases to 120 bpm confirming the functionality of every rate adaptive mode. This test also confirms the second requirement for the rate adaptive modes indicating that the pacing rate should increase under activity.

Continuing the above test, we proceeded to rest the Pacemaker. It is expected that the heart rate should return to 60 BPM, our Lower Rate Limit.

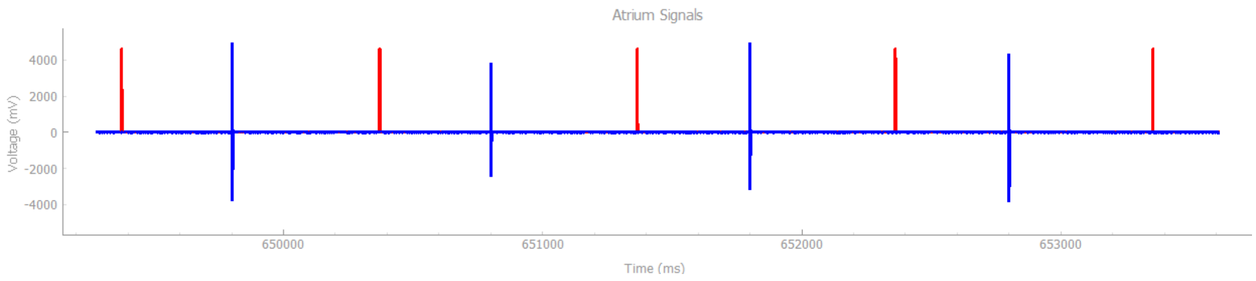


Figure 34: AOOR returning to 60 bpm after no activity

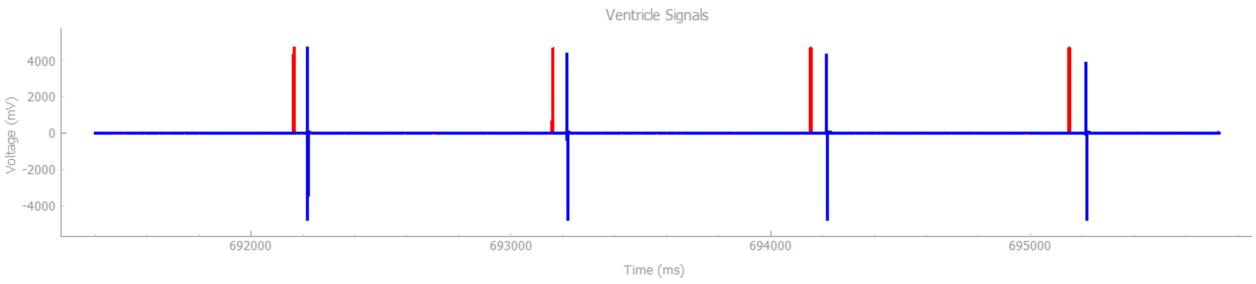


Figure 35: VOOR returning to 60 bpm after no activity

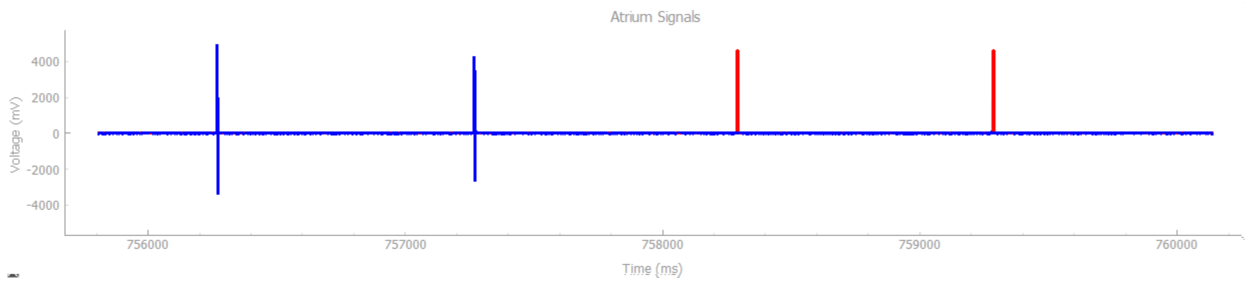


Figure 36: AAIR returning to 60 bpm and pacing until natural pace

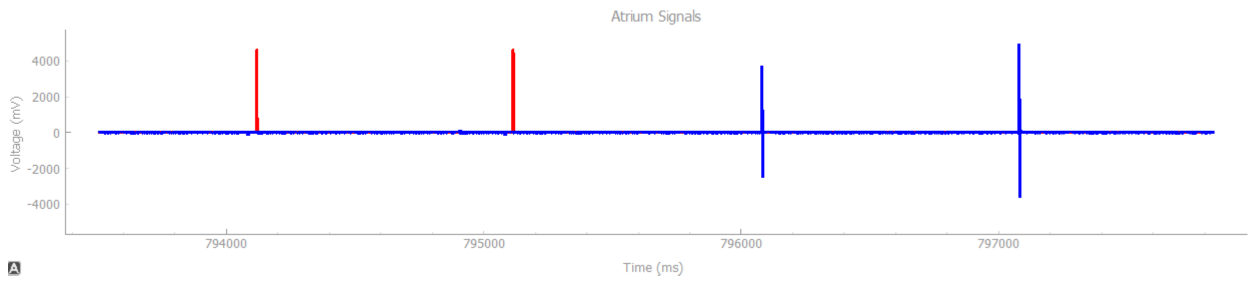


Figure 37: AAIR returning to 60 bpm and pacing once no natural pace sensed

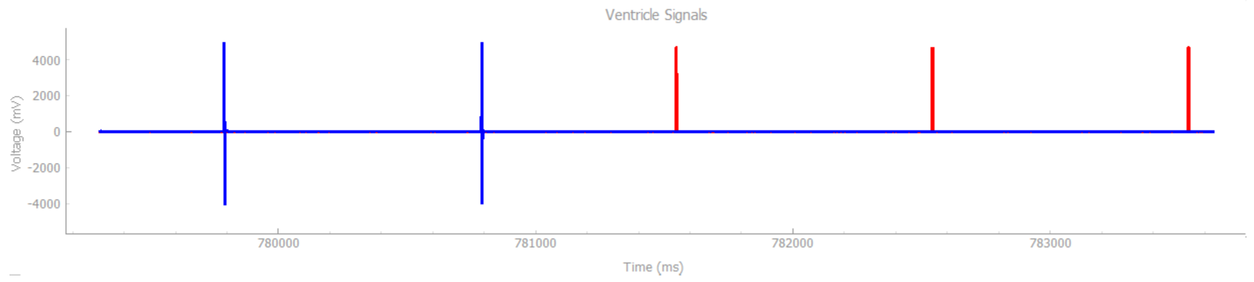


Figure 38: VVIR returning to 60 bpm and stops pacing once natural pace sensed

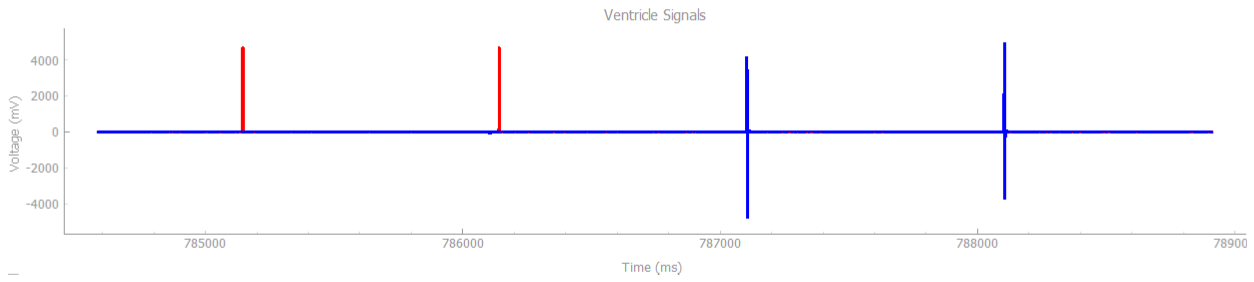


Figure 39: VVIR returning to 60 bpm and starts pacing when no natural pace sensed

After our set recovery time (30 seconds), all modes paced at a rate of 60 BPM confirming that each rate adaptive mode is functioning as expected after rest. This test also confirms the final requirement for all rate adaptive modes indicating that they should return to the lower rate limit after the set recovery time.

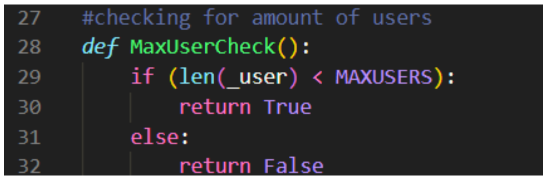
1.4.2 DCM Testing

For testing the DCM, we needed to verify for when 10 users had been registered, an incorrect and correct username and an incorrect and correct password. We also wanted to test if our modification of parameters worked and whether the JSON file saved data when closing the app. Finally, we needed to test if the parameter values were being properly sent to the pacemaker. We implemented output messages through the tkinter library to display when an incorrect password, username and registration at 10 users occurred. We could then check the JSON file to verify each of these tests. We started by filling the JSON file with 10 users to test our max user function shown in Figure 40. The maxUserCheck worked as intended and gave us the correct output message (Figure 41) and did not append any new users to the userlist in the JSON file.

For the username and password checking we implemented several functions to compare the inputted usernames and passwords to already stored values. When an existing username was registered, we wanted to display a message saying the username is already taken. This worked as intended, seen in Figure 42. We also tested that the user login worked correctly and displayed an "incorrect username or password" message. In Figure 43 the correct output is seen, working for both an incorrect password and username. We also testing this with inputting a correct password for a different user and that also passed the test, giving the same output message. Our last test was with our json file and parameter modification.

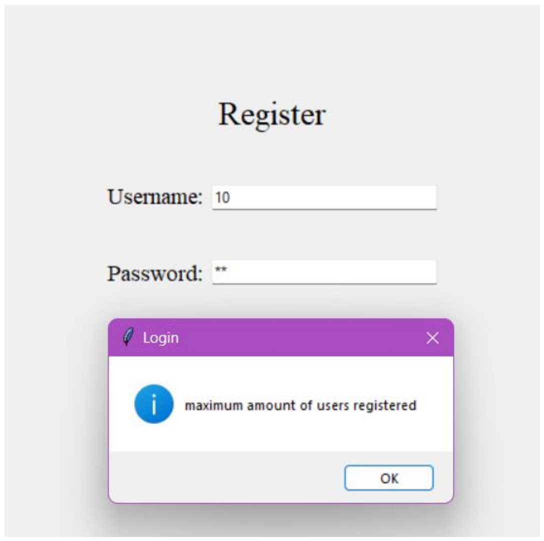
We tested if the parameter values reset or did not save when closing the app or leaving the specific page. This worked as intended as when reloading the parameter values in our pacing mode screens all parameter values were the same as when they were last modified. We also tested that the parameters only changed when the submit button was pressed so that parameters could not be changed by accidently moving the sliders. In Figure 44, the first lower rate limit value has changed from its nominal value 60ppm to 133ppm and has then been updated in the JSON file.

We tested our UART functions by sending the data from the DCM and printing it to the terminal. We also used an LED on the Pacemaker that turned on when UART data was received. With this we could verify that the correct data is sent and that the Simulink code received that data. In Figure 45 the print for the UART data is shown.



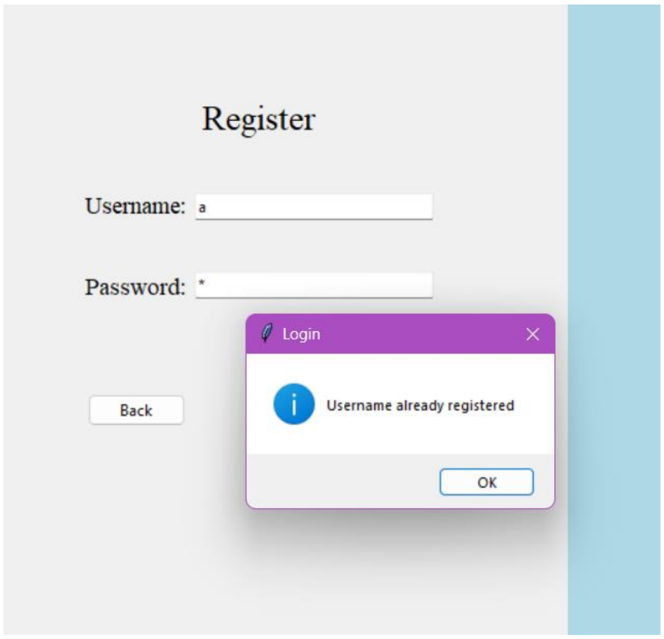
```python
#checking for amount of users
def MaxUserCheck():
    if (len(_user) < MAXUSERS):
        return True
    else:
        return False
```

Figure 40. Check for Maximum Users



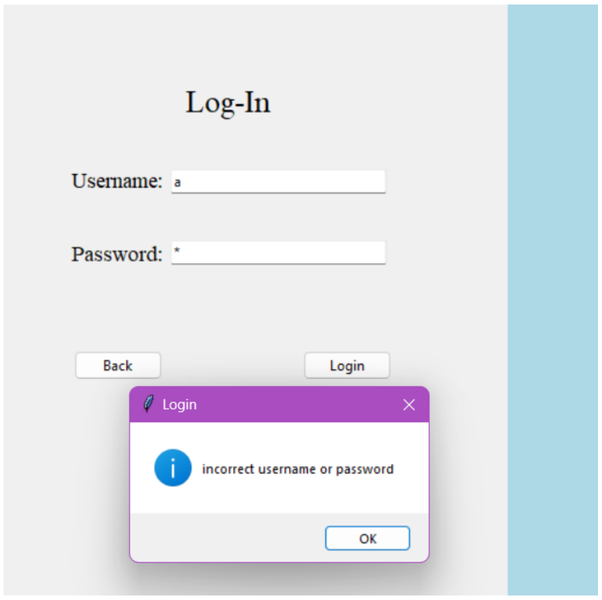
maximum amount of users registered

Figure 41. Max User Message



Username already registered

Figure 42: Already Existing Username Message


Log-In

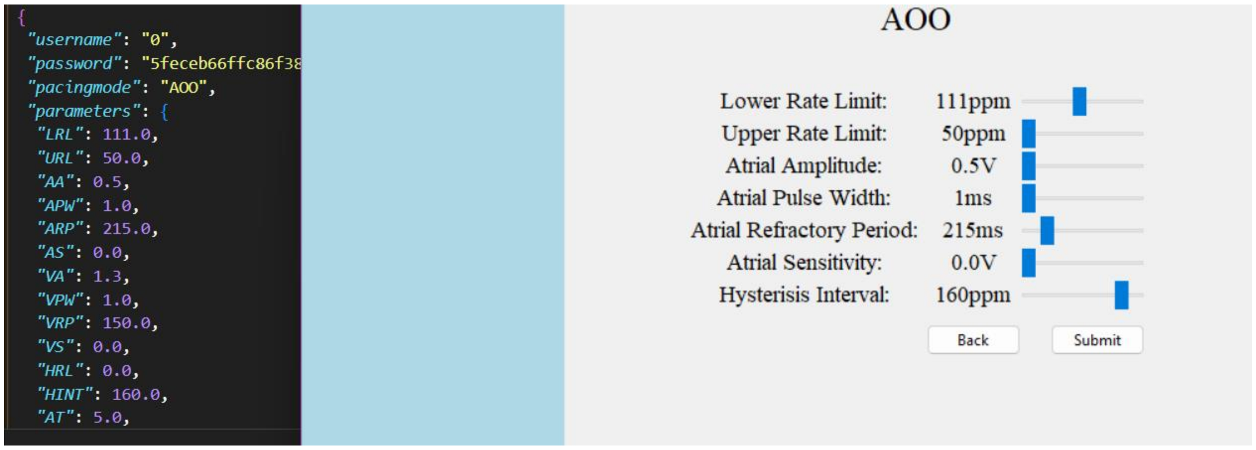
Username: a

Password: *

Login

incorrect username or password

Figure 43: Incorrect Username/Password

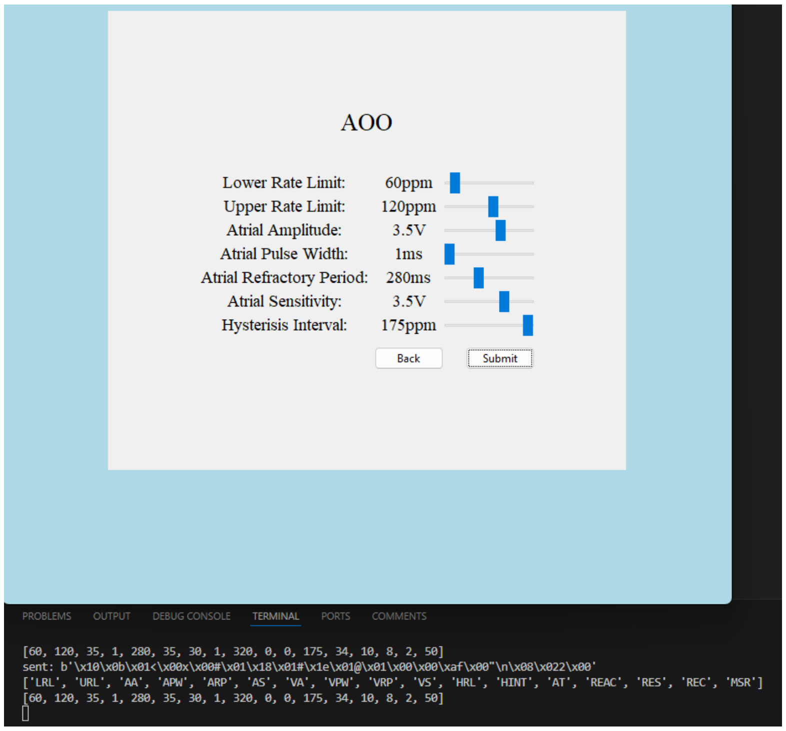


```
{
 "username": "0",
 "password": "5feceb66ffc86f38
 "pacingmode": "AOO",
 "parameters": {
  "LRL": 111.0,
  "URL": 50.0,
  "AA": 0.5,
  "APW": 1.0,
  "ARP": 215.0,
  "AS": 0.0,
  "VA": 1.3,
  "VPW": 1.0,
  "VRP": 150.0,
  "VS": 0.0,
  "HRL": 0.0,
  "HINT": 160.0,
  "AT": 5.0,
```

AOO

Lower Rate Limit:	111ppm
Upper Rate Limit:	50ppm
Atrial Amplitude:	0.5V
Atrial Pulse Width:	1ms
Atrial Refractory Period:	215ms
Atrial Sensitivity:	0.0V
Hysterisis Interval:	160ppm

Figure 44: Json and Modified Parameters



Lower Rate Limit:	60ppm	
Upper Rate Limit:	120ppm	
Atrial Amplitude:	3.5V	
Atrial Pulse Width:	1ms	
Atrial Refractory Period:	280ms	
Atrial Sensitivity:	3.5V	
Hysterisis Interval:	175ppm	

```
[60, 120, 35, 1, 280, 35, 30, 1, 320, 0, 0, 175, 34, 10, 8, 2, 50]
sent: b'\x10\x0b\x01<\x00x\x00#\x01\x18\x01#\x1e\x01@\x01\x00\x00\xaf\x00"\n\x08\x022\x00'
['LRL', 'URL', 'AA', 'APW', 'ARP', 'AS', 'VA', 'VPW', 'VRP', 'VS', 'HRL', 'HINT', 'AT', 'REAC', 'RES', 'REC', 'MSR']
[60, 120, 35, 1, 280, 35, 30, 1, 320, 0, 0, 175, 34, 10, 8, 2, 50]
```

Figure 45: UART test with terminal

2. Future Requirements and Modules

2.1 Future Requirements

2.1.1 Simulink

2.1.1.1 Requirements likely to change

In the future dual pacing modes could be added to the pacemaker state diagram (VDD, DDI, DDD) to enhance functionality, and the pin assignment subsystem will be updated accordingly to support these new modes.

2.1.1.2 Design decisions likely to change

Improvements to the serial communication could be made to allow for more consistent transfer of data between the pacemaker and the DCM.

If more pacing modes are added, the algorithm to calculate rate adaptive pacing could also be updated to accommodate.

2.1.2 DCM

2.1.2.1 Requirements likely to change

A requirement likely to be added is the function to delete a user. Right now, the user cannot remove their account from the app. Deleting a user is necessary for the DCM and will help in testing the app. In the future we will implement a way for the user to remove their own account and delete all their data from the JSON storage file. Currently, Egram data is being graphed but the UART information we are receiving does not properly get transferred to the graph. In the future we need to refine how the received data is interpreted so the Egram is properly displayed for the user. Another requirement needed to change is adding the entire list of parameters. Assignment one and two only required half of the parameters and pacing modes to be modifiable on the DCM. We will need to add the remaining parameters and have them be modifiable just like the parameters implemented now. Another requirement that we are likely to change in the future is putting a status light that determines connectivity to the pacemaker. Currently, it only displays in text by showing the serial number of the connected device. Adding a light will make it more recognizable if the device is connected or not.

2.1.2.2 Design decisions likely to change

An important design decision we will likely change in the future will be the implementation of the JSON data to have more way to hide or protect user information. Right now, the data is stored in the JSON file and Sha256 hashing is implemented to hash the password and make it slightly more secure. Instead, we could use a different library from JSON and implement a more secure data storing method to hash or encrypt the entirety of the user's data instead of just their passwords. We currently use the Pyserial library so we can have a UART connection with the Pacemaker. This connection works sending the data to the Pacemaker but the receiving part of this for the Egram data is not fully implemented. The Egram is received very slowly and not always accurate, in the future we could look at different ways of receiving the UART data so the system can run smoother and faster than our current implementation. With our current method to modify parameters we have a submit button, so that the parameter values have one more layer of security from being changed by accident. To make this more secure we could add a password input so that only authorized users who know their password are able to submit and change the parameter values. The sliders also currently do not support incremental changes exactly how the increments are shown in the Pacemaker documentation. To fix this we can investigate different slider methods or ways to increment the values so that they properly match the documentation.

2.2 Module Interface Specification

Describes the purpose, public functions, and black-box behaviour of each function

2.2.1 Simulink

2.2.1.1 Inputs and Outputs Subsystem

Inputs	- PACING_REF_PWM - ATR_CMP_REF_PWM - VENT_CMP_REF_PWM - PACE_CHARGE_CTRL - PACE_GND_CTRL - ATR_PACE_CTRL - ATR_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL

	- FRONTEND_CTRL - Z_ATR_CTRL - Z_VENT_CTRL - LED_ON
Outputs	- ATR_CMP_DETECT - VENT_CMP_DETECT - ACTIVITY_LEVEL - LED_ON1 - LED_ON2 - VENT_SIGNAL - ATR_SIGNAL
Behavior	Communicates between the Simulink state flow and the pacemaker board, allowing hardware hiding by allocating inputs to corresponding pins. Calculates ACTIVITY_LEVEL based on accelerometer data.

Table 2: Inputs and Outputs to the MATLAB Subsystem

2.2.1.2 Main stateflow

Inputs	- State_Chosen - Atrial_Pulse_Width - Ventricle_Pulse_Width - Atrial_Amplitude - Ventricle_Amplitude - VRP - ARP - Lower_Rate_Limit - Upper_Rate_Limit - Hysteresis_Interval - ATR_CMP_REF_PWM - VENT_CMP_REF_PWM - ATR_CMP_DETECT - VENT_CMP_DETECT
Outputs	- PACE_CHARGE_CTRL - PACE_GND_CTRL - ATR_PACE_CTRL - ATR_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - FRONTEND_CTRL - LED_ON - Z_ATR_CTRL - Z_VENT_CTRL
Behavior	Allows the program to switch between different modes, organizing smaller state flows and making it easy to add more modes in the future. This setup keeps things organized and ready for future improvements.

Table 3: Inputs and Outputs to the Main State Flow

2.2.1.3 AOO stateflow

Inputs	- Atrial_Pulse_Width - Atrial_Amplitude - Lower_Rate_Limit
Outputs	- PACE_CHARGE_CTRL - PACE_GND_CTRL - ATR_PACE_CTRL - ATR_GND_CTRL - Z_ATR_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_VENT_CTRL
Behavior	Implements AOO pacemaker mode by using programmable parameters like the lower rate limit and atrial pulse width. These control when and how the pacing pulses are sent.

Table 4: Inputs and Outputs to the AOO State Flow

2.2.1.4 VOO stateflow

Inputs	- Ventricle_Pulse_Width - Ventricle_Amplitude - Lower_Rate_Limit
Outputs	- PACE_CHARGE_CTRL - PACE_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_VENT_CTRL - ATR_PACE_CTRL - ATR_GND_CTRL - Z_ATR_CTRL
Behavior	Implements VOO pacemaker mode by using programmable parameters like the lower rate limit and atrial pulse width. These control when and how the pacing pulses are sent.

Table 5: Inputs and Outputs to the VOO State Flow

2.2.1.5 AAI stateflow

Inputs	- Atrial_Pulse_Width - Atrial_Amplitude - ARP - Lower_Rate_Limit - Hysteresis_Interval - ATR_CMP_REF_PWM_IN - ATR_CMP_DETECT

Outputs	- PACING_REF_PWM - ATR_CMP_REF_PWM - FRONTEND_CTRL - PACE_CHARGE_CTRL - ATR_PACE_CTRL - PACE_GND_CTRL - ATR_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_ATR_CTRL - Z_VENT_CTRL
Behavior	Implements AAI mode through using parameters including FRONTEND_CTRL which allows the pacemaker to sense atrial activity to detect when a heartbeat occurs. Parameters such as PACE_CHARGE_CTRL is needed for pacing when in a state where sensing is disabled.

Table 6: Inputs and Outputs to the AAI State Flow

2.2.1.6 VVI stateflow

Inputs	- Ventricle_Pulse_Width - Ventricle_Amplitude - VRP - Lower_Rate_Limit - Hysteresis_Interval - VENT_CMP_REF_PWM_IN - VENT_CMP_DETECT
Outputs	- PACING_REF_PWM - VENT_CMP_REF_PWM - FRONTEND_CTRL - PACE_CHARGE_CTRL - ATR_PACE_CTRL - PACE_GND_CTRL - ATR_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_ATR_CTRL - Z_VENT_CTRL
Behavior	Implements VVI mode through using parameters which allows the pacemaker to sense ventricle activity to detect when a heartbeat occurs and parameters which are needed for pacing when in a state where sensing is disabled.

Table 7: Inputs and Outputs to the VVI State Flow

2.2.1.7 AOOR stateflow

Inputs	- Atrial_Pulse_Width - Atrial_Amplitude - Lower_Rate_Limit
Outputs	- PACE_CHARGE_CTRL - PACE_GND_CTRL

	- ATR_PACE_CTRL - ATR_GND_CTRL - Z_ATR_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_VENT_CTRL
Behavior	Implements AOOR pacemaker mode by using programmable parameters like the atrial pulse width and variable parameters like the CPR. These control when and how the pacing pulses are sent.

Table 8: Inputs and Outputs to the AOOR Stateflow

2.2.1.8 VOOR stateflow

Inputs	- Ventricle_Pulse_Width - Ventricle_Amplitude - CPR
Outputs	- PACE_CHARGE_CTRL - PACE_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_VENT_CTRL - ATR_PACE_CTRL - ATR_GND_CTRL - Z_ATR_CTRL
Behavior	Implements VOOR pacemaker mode by using programmable parameters like the ventricle pulse width and variable parameters like the CPR. These control when and how the pacing pulses are sent.

Table 9: Inputs and Outputs to the VOOR Stateflow

2.2.1.9 AAIR stateflow

Inputs	- Atrial_Pulse_Width - Atrial_Amplitude - ARP - CPR - Hysteresis_Interval - ATR_CMP_REF_PWM_IN - ATR_CMP_DETECT
Outputs	- PACING_REF_PWM - ATR_CMP_REF_PWM - FRONTEND_CTRL - PACE_CHARGE_CTRL - ATR_PACE_CTRL - PACE_GND_CTRL - ATR_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_ATR_CTRL - Z_VENT_CTRL

Behavior	Implements AAIR mode through using parameters including FRONTEND_CTRL which allows the pacemaker to sense atrial activity to detect when a heartbeat occurs. Parameters such as PACE_CHARGE_CTRL is needed for pacing when in a state where sensing is disabled.

Table 10: Inputs and Outputs to the AAIR Stateflow

2.2.1.10 VVIR stateflow

Inputs	- Ventricle_Pulse_Width - Ventricle_Amplitude - VRP - CPR - Hysteresis_Interval - VENT_CMP_REF_PWM_IN - VENT_CMP_DETECT
Outputs	- PACING_REF_PWM - VENT_CMP_REF_PWM - FRONTEND_CTRL - PACE_CHARGE_CTRL - ATR_PACE_CTRL - PACE_GND_CTRL - ATR_GND_CTRL - VENT_PACE_CTRL - VENT_GND_CTRL - Z_ATR_CTRL - Z_VENT_CTRL
Behavior	Implements VVIR mode through using parameters which allows the pacemaker to sense ventricle activity to detect when a heartbeat occurs and parameters which are needed for pacing when in a state where sensing is disabled.

Table 11: Inputs and Outputs to the VVIR Stateflow

2.2.1.11 Serial transmit to/from DCM Subsystem

Inputs	- VENT_SIGNAL - ATR_SIGNAL
Outputs	- Response_Factor - Activity_Threshold - reaction_Time - Max_Sensor_Rate - PacingMode - Lower_Rate_Limit - Upper_Rate_Limit - Atrial_Amplitude - Atrial_Pulse_Width - ARP - ATR_CMP_REF_PWM - Ventricular_Amplitude - Ventricular_Pulse_Width

	- VRP - VENT_CMP_REF_PWM - Hysteresis_Interval - Hysteresis
Behavior	Holds the state flow that receives data from DCM and the subsystem that transmits data to DCM

Table 12: Inputs and Outputs for sending and receiving UART

2.2.1.12 Receiving Data from DCM Stateflow

Inputs	- rxdata - status
Outputs	- FnCode - PacingMode - Lower_Rate_Limit - Upper_Rate_Limit - Atrial_Amplitude - Atrial_Pulse_Width - ARP - ATR_CMP_REF_PWM - hysteresis - Hysteresis_Interval - Activity_Threshold - Reaction_Time - Response_Factor - Recovrey_Time - Max_Sensor_Rate
Behavior	Manages the operation of receiving data through UART, setting parameters and transitioning between states based on the data received

Table 13: Inputs and Outputs for UART from DCM

2.2.1.13 Transmitting Data to DCM through UART subsystem

Inputs	- FnCode - PacingMode - Lower_Rate_Limit - Upper_Rate_Limit - Atrial_Amplitude - Atrial_Pulse_Width - ARP - ATR_CMP_REF_PWM - hysteresis - Hysteresis_Interval - Activity_Threshold - Reaction_Time - Response_Factor - Recovrey_Time - Max_Sensor_Rate - ATR_SIGNAL

	- VENT_SIGNAL
Outputs	No Outputs
Behavior	The send_uart function is called here and data is sent to DCM through UART.

Table 14: Inputs and Outputs for UART data transmission

2.2.1.14 DPR Subsystem

Inputs	- RESPONSE_FACTOR - ACTIVITY_LEVEL - ACTIVITY_THRESHOLD - LRL
Outputs	- DPR
Behavior	Calculates DPR based on the above inputs.

Table 15: DPR System

2.2.1.15 Current Pacing Rate Stateflow

Inputs	- DPR - REACTION_TIME - RECOVERY_TIME - MAX_SENSOR_RATE - LRL
Outputs	- LEDON1 - LEDON2 - NCPR
Behavior	Increases and decreases the CPR based on the provided inputs. Also changes the LED colour based on if it’s increasing or decreasing.

Table 16: Pacing Rate Stateflow

2.2.2 DCM

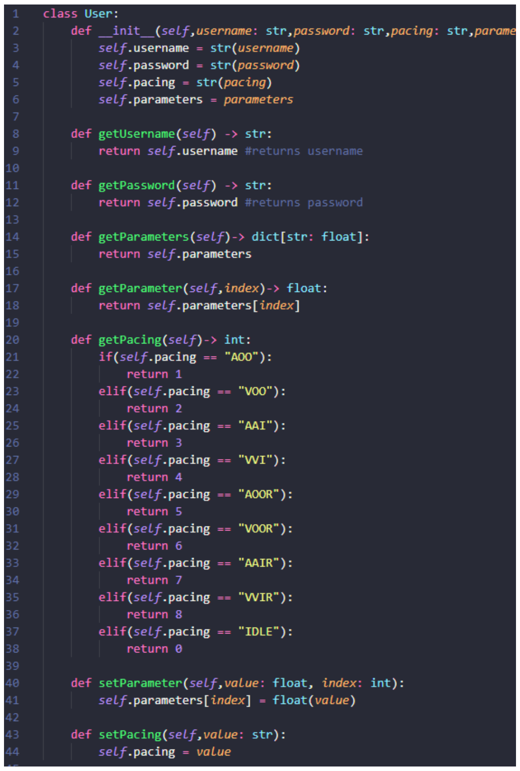
2.2.2.1 User Module

The User module is the data structure used to house all the users’ values. Password, username, parameters and pacing modes are all stored here so they can all be called and saved in the json file.

Function	**Purpose**	**Parameters**
getUsername()	Returning the username value of the user.	self
getPassword()	Returning the password value of the user .	self
getParameters()	Returns the entire dictionary of parameters.	self
getParameter()	Returns a specific parameter value at the given index.	Self,index
getPacing()	Returns the current pacing mode but as a integer instead of as a string, values are	self

	returned as integers so when sent to the Pacemaker it is easier to change what pacing mode to use.	
setParameter()	For the parameter at the given index, updates its value with the given value.	Self,value,index
setPacing()	For the pacing mode of the user, updates it if a different pacing mode is selected.	Self,value

Table 17: User Module Functions



```python
class User:
    def __init__(self,username: str,password: str,pacing: str,parame
        self.username = str(username)
        self.password = str(password)
        self.pacing = str(pacing)
        self.parameters = parameters

    def getUsername(self) -> str:
        return self.username #returns username

    def getPassword(self) -> str:
        return self.password #returns password

    def getParameters(self)-> dict[str: float]:
        return self.parameters

    def getParameter(self,index)-> float:
        return self.parameters[index]

    def getPacing(self)-> int:
        if(self.pacing == "AOO"):
            return 1
        elif(self.pacing == "VOO"):
            return 2
        elif(self.pacing == "AAI"):
            return 3
        elif(self.pacing == "VVI"):
            return 4
        elif(self.pacing == "AOOR"):
            return 5
        elif(self.pacing == "VOOR"):
            return 6
        elif(self.pacing == "AAIR"):
            return 7
        elif(self.pacing == "VVIR"):
            return 8
        elif(self.pacing == "IDLE"):
            return 0

    def setParameter(self,value: float, index: int):
        self.parameters[index] = float(value)

    def setPacing(self,value: str):
        self.pacing = value
```

Figure 46: User Module

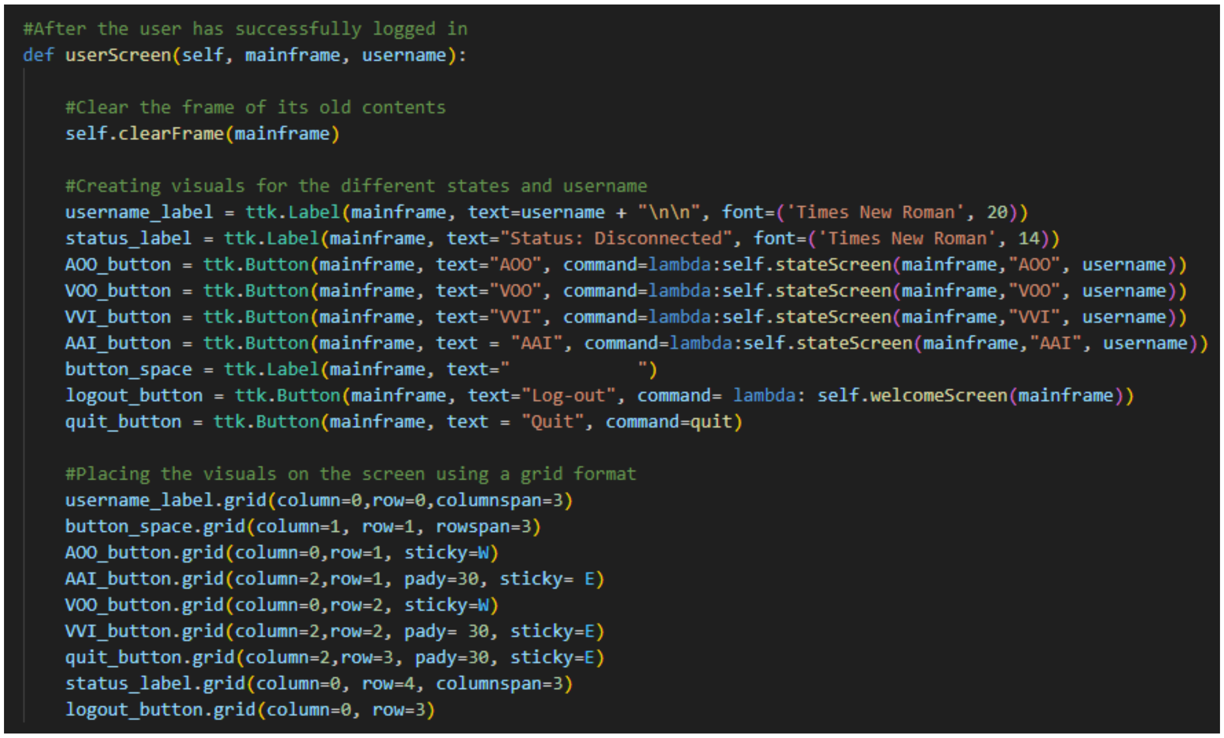
2.2.2.2 Interface Module

The interface module is responsible for the visuals within the DCM. Each function ending with “screen” represents a different screen within the DCM. The clearFrame() function is called at the beginning of each function to clear the frame of its existing contents from the previous page it was on. The update functions update the user-specific parameter values and store them in the JSON file containing user data. There were new functions added such as egramScreen() which makes a new window showing the egram data that would be sent from the pacemaker to be displayed to the user.

Function	Purpose	Parameters
loginScreen()	Displays the login screen with entry fields for username and password and a login button	Self, mainframe

registerScreen()	Displays the registration screen with entry fields for username and password and register button	Self, mainframe
welcomeScreen()	Displays welcome screen with Log-In, Register, and quit button, as well as a status label indicating whether the pacemaker is connected.	Self, mainframe
login()	Checks the login credentials entered by the user, and returns a message box to the user accordingly	Self, mainframe
userScreen()	Displays the user screen with pacing mode options and a log-out button.	Self, mainframe, username
stateScreen()	Shows parameter settings based on selected pacing mode, with sliders to adjust values.	Self, mainframe, state name, username
SubmitPressed()	Raises flag to update slider value	Self, username
UpdateLabel()	Updates the label when the user is changing the sliders, does not change the parameter information in the JSON file but updates only visually for the user.	(self, value, key, unit)
updateSliderValue()	Updates the slider value on the screen and adjusts the parameter value for the user.	Self, username, key, value
updateParameter()	Updates the parameter on the screen and stores the value in a JSON file.	Self, Parameter object, value, label, username
register()	Registers a new user by checking credentials and displaying a message on success or failure.	Self
clearFrame()	Removes all widgets from the specified frame.	Self, frame
egramScreen()	Displays real-time egram data as a scrolling plot using Matplotlib	Self, mainframe

Table 18: Interface.py file function specification



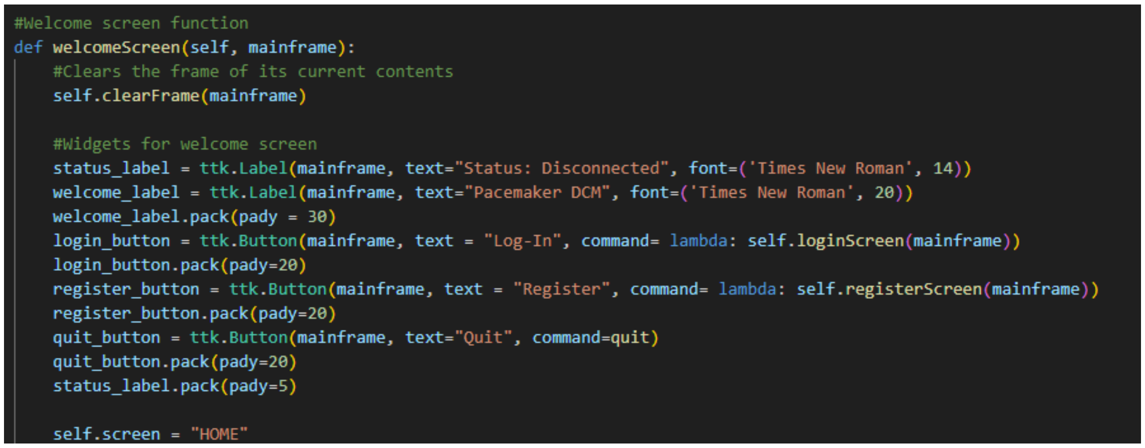
```
#After the user has successfully logged in
def userScreen(self, mainframe, username):

    #Clear the frame of its old contents
    self.clearFrame(mainframe)

    #Creating visuals for the different states and username
    username_label = ttk.Label(mainframe, text=username + "\n\n", font=('Times New Roman', 20))
    status_label = ttk.Label(mainframe, text="Status: Disconnected", font=('Times New Roman', 14))
    AOO_button = ttk.Button(mainframe, text="AOO", command=lambda:self.stateScreen(mainframe,"AOO", username))
    VOO_button = ttk.Button(mainframe, text="VOO", command=lambda:self.stateScreen(mainframe,"VOO", username))
    VVI_button = ttk.Button(mainframe, text="VVI", command=lambda:self.stateScreen(mainframe,"VVI", username))
    AAI_button = ttk.Button(mainframe, text = "AAI", command=lambda:self.stateScreen(mainframe,"AAI", username))
    button_space = ttk.Label(mainframe, text="            ")
    logout_button = ttk.Button(mainframe, text="Log-out", command= lambda: self.welcomeScreen(mainframe))
    quit_button = ttk.Button(mainframe, text = "Quit", command=quit)

    #Placing the visuals on the screen using a grid format
    username_label.grid(column=0,row=0,columnspan=3)
    button_space.grid(column=1, row=1, rowspan=3)
    AOO_button.grid(column=0,row=1, sticky=W)
    AAI_button.grid(column=2,row=1, pady=30, sticky= E)
    VOO_button.grid(column=0,row=2, sticky=W)
    VVI_button.grid(column=2,row=2, pady= 30, sticky=E)
    quit_button.grid(column=2,row=3, pady=30, sticky=E)
    status_label.grid(column=0, row=4, columnspan=3)
    logout_button.grid(column=0, row=3)
```

Figure 47: User Screen function



```
#Welcome screen function
def welcomeScreen(self, mainframe):
    #Clears the frame of its current contents
    self.clearFrame(mainframe)

    #Widgets for welcome screen
    status_label = ttk.Label(mainframe, text="Status: Disconnected", font=('Times New Roman', 14))
    welcome_label = ttk.Label(mainframe, text="Pacemaker DCM", font=('Times New Roman', 20))
    welcome_label.pack(pady = 30)
    login_button = ttk.Button(mainframe, text = "Log-In", command= lambda: self.loginScreen(mainframe))
    login_button.pack(pady=20)
    register_button = ttk.Button(mainframe, text = "Register", command= lambda: self.registerScreen(mainframe))
    register_button.pack(pady=20)
    quit_button = ttk.Button(mainframe, text="Quit", command=quit)
    quit_button.pack(pady=20)
    status_label.pack(pady=5)

    self.screen = "HOME"
```

Figure 48: Welcome Screen function

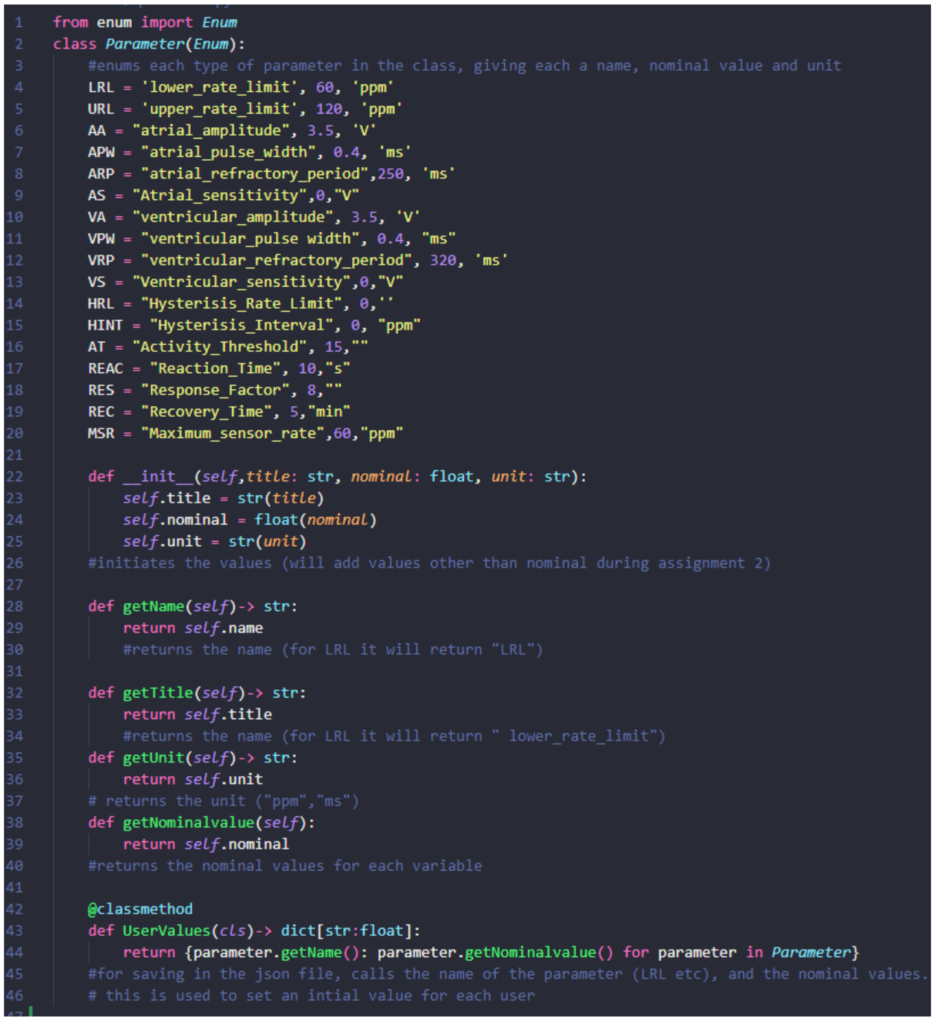
2.2.2.3 Parameter Module

The parameter module holds all the call values for the parameter, it holds all the nominal values, units and names of the parameter so that they can be inputted into the users parameters and then saved to json.

Function	**Purpose**	**Parameters**
getName()	Returns the name of the parameter, used to check	self

	which parameter is being updated	
getTitle()	Returns the title of the parameter, instead of name the title is the long form string name of the parameter, instead of the short form	self
getUnit()	Returns the unit of the parameter as a string	self
getNominalValue()	Returns the nominal value of the parameter	self
userValues()	Returns the values that are going to be saved in the json file, is in a for loop so that every parameter is outputted	class

Table 19: Parameter Module Functions



```python
from enum import Enum
class Parameter(Enum):
    #enums each type of parameter in the class, giving each a name, nominal value and unit
    LRL = 'lower_rate_limit', 60, 'ppm'
    URL = 'upper_rate_limit', 120, 'ppm'
    AA = "atrial_amplitude", 3.5, 'V'
    APW = "atrial_pulse_width", 0.4, 'ms'
    ARP = "atrial_refractory_period",250, 'ms'
    AS = "Atrial_sensitivity",0,"V"
    VA = "ventricular_amplitude", 3.5, 'V'
    VPW = "ventricular_pulse width", 0.4, "ms"
    VRP = "ventricular_refractory_period", 320, 'ms'
    VS = "Ventricular_sensitivity",0,"V"
    HRL = "Hysterisis_Rate_Limit", 0,''
    HINT = "Hysterisis_Interval", 0, "ppm"
    AT = "Activity_Threshold", 15,""
    REAC = "Reaction_Time", 10,"s"
    RES = "Response_Factor", 8,""
    REC = "Recovery_Time", 5,"min"
    MSR = "Maximum_sensor_rate",60,"ppm"

    def __init__(self,title: str, nominal: float, unit: str):
        self.title = str(title)
        self.nominal = float(nominal)
        self.unit = str(unit)
    #initiates the values (will add values other than nominal during assignment 2)

    def getName(self)-> str:
        return self.name
        #returns the name (for LRL it will return "LRL")

    def getTitle(self)-> str:
        return self.title
        #returns the name (for LRL it will return " lower_rate_limit")
    def getUnit(self)-> str:
        return self.unit
    # returns the unit ("ppm","ms")
    def getNominalvalue(self):
        return self.nominal
    #returns the nominal values for each variable

    @classmethod
    def UserValues(cls)-> dict[str:float]:
        return {parameter.getName(): parameter.getNominalvalue() for parameter in Parameter}
    #for saving in the json file, calls the name of the parameter (LRL etc), and the nominal values.
    # this is used to set an intial value for each user
```

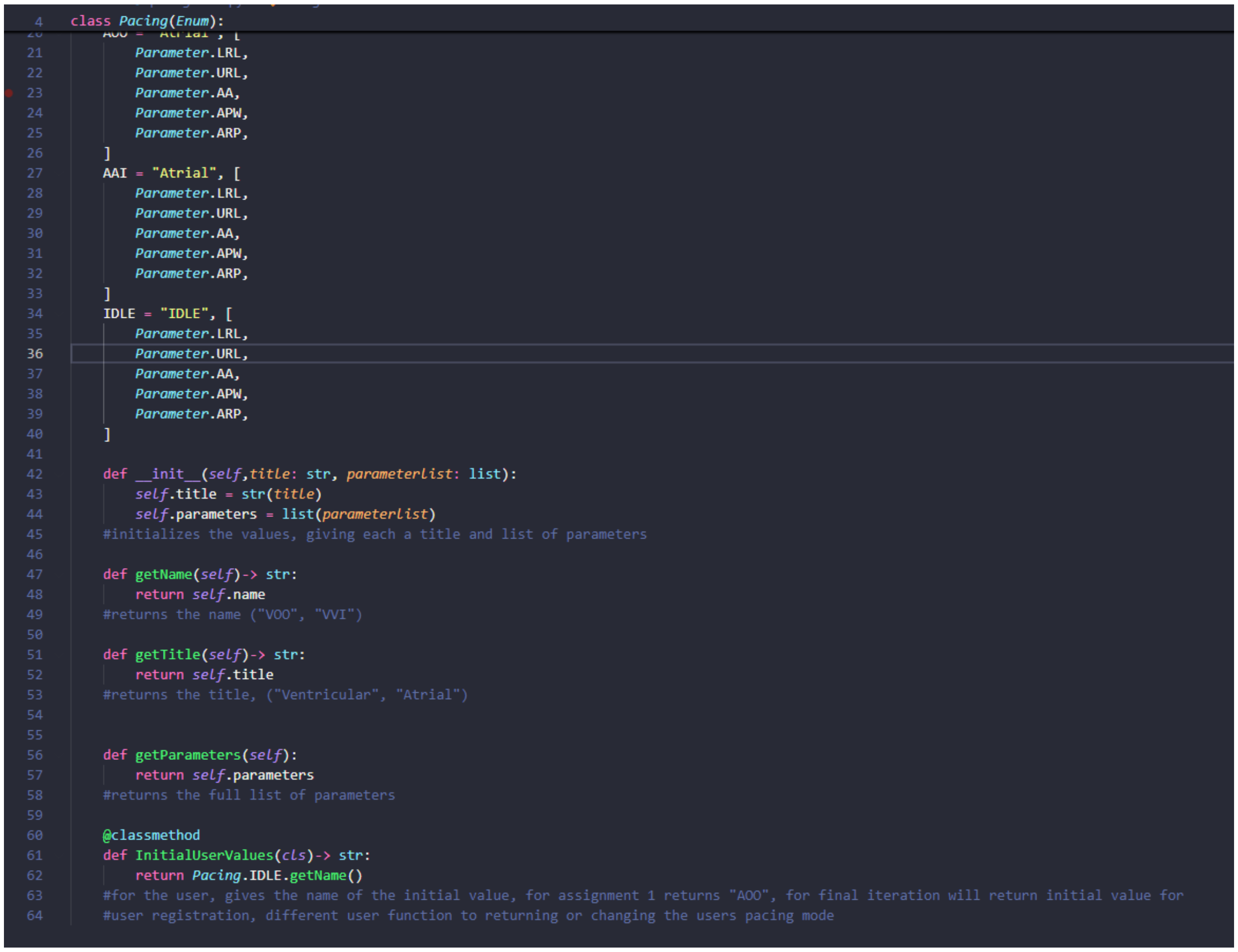
Figure 49. Parameter Module

2.2.2.4 Pacing mode Module

The pacing mode module holds all the functions to call the pacing mode values, this is used to set the users pacing mode value to be saved in the json file.

Function	**Purpose**	**Parameters**
getName()	Returns the name of the pacing mode	self
getTitle()	Returns the title of the pacing mode	Self
getParameters()	Returns the list of parameters needed for each parameter	Self
InitialUserValues()	Returns a default initial value for the user	Class

Table 20: Pacing Mode Module Functions



```python
class Pacing(Enum):
    AOO = "Atrial", [
        Parameter.LRL,
        Parameter.URL,
        Parameter.AA,
        Parameter.APW,
        Parameter.ARP,
    ]
    AAI = "Atrial", [
        Parameter.LRL,
        Parameter.URL,
        Parameter.AA,
        Parameter.APW,
        Parameter.ARP,
    ]
    IDLE = "IDLE", [
        Parameter.LRL,
        Parameter.URL,
        Parameter.AA,
        Parameter.APW,
        Parameter.ARP,
    ]

    def __init__(self,title: str, parameterlist: list):
        self.title = str(title)
        self.parameters = list(parameterlist)
    #initializes the values, giving each a title and list of parameters

    def getName(self)-> str:
        return self.name
    #returns the name ("VOO", "VVI")

    def getTitle(self)-> str:
        return self.title
    #returns the title, ("Ventricular", "Atrial")


    def getParameters(self):
        return self.parameters
    #returns the full list of parameters

    @classmethod
    def InitialUserValues(cls)-> str:
        return Pacing.IDLE.getName()
    #for the user, gives the name of the initial value, for assignment 1 returns "AOO", for final iteration will return initial value for
    #user registration, different user function to returning or changing the users pacing mode
```

Figure 50. Pacing Mode Module

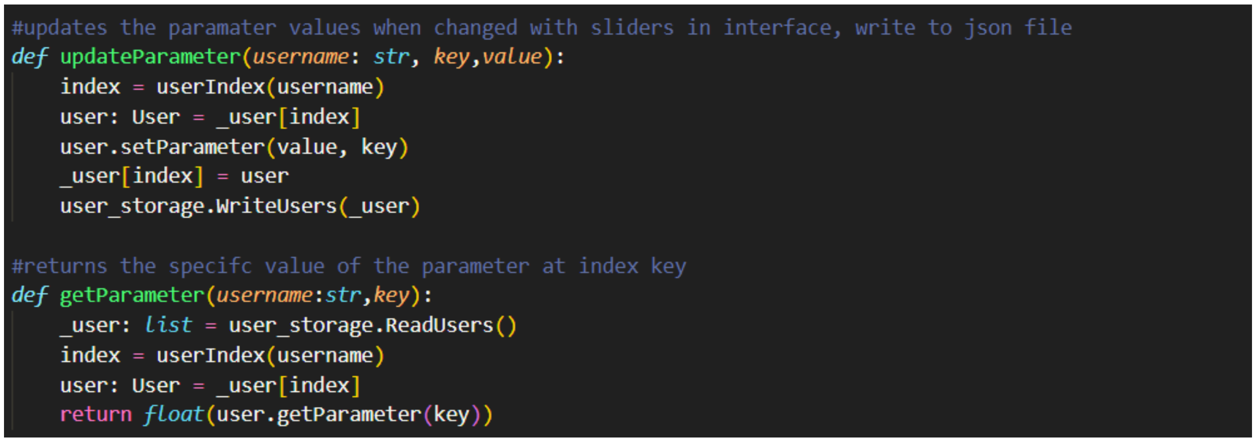
2.2.2.5 User_Processing Module

The User_Processing module is used to facilitate all checking of login, registration and changing of parameters. The functions in the module check what data has been inputted and confirms if it can be saved to the json file or is correct for logging in. User_processing also pulls all the user’s parameters from the json file so it can be sent in the UART module.

Function	**Purpose**	**Parameter**
Registercheck()	Creates a new user using the user class and then checks if the user already exists, makes sure there are not 10 users and then appends the user to the list of users to save into the json file. Sends a error message to the interface if user exists or if max users amount has been reached	Username,password
Logincheck()	Checks if the new user doesn’t already exist, and then checks if the password is correct. if both username and password is correct then sends message to interface, otherwise send error message to interface.	Username,password
maxUserCheck()	Checks if the length of the user list is less than 10, returns true if it is false if its over 10	
checkUser()	Checks if the user already exists by going through each user in the list and checking if the username is the same as the inputed one	User
userIndex()	Returns the index of the user in the list	username
CheckPassword()	Checks if the inputted user has the same password as the password at the index given.	User,index
updateParameter()	Takes the username to find the specific index in the list to update the user, takes the key and calls the user.setvalue() to change its value, then rewrites to storage to save to the json file	Username,key,value
getParameter()	Reads the json to make sure its all up to date, takes the username to find the index value and then returns the parameter at the key value.	Username,key

UpdatePacing()	Takes the username to find index so that pacing mode can be updated to its new value for the specific user.	Username, value
getPacing()	Reads the json file and returns the current pacing mode for the specified user.	username

Table 21: User Processing Module Functions



```
#updates the paramater values when changed with sliders in interface, write to json file
def updateParameter(username: str, key,value):
    index = userIndex(username)
    user: User = _user[index]
    user.setParameter(value, key)
    _user[index] = user
    user_storage.WriteUsers(_user)

#returns the specifc value of the parameter at index key
def getParameter(username:str,key):
    _user: list = user_storage.ReadUsers()
    index = userIndex(username)
    user: User = _user[index]
    return float(user.getParameter(key))
```

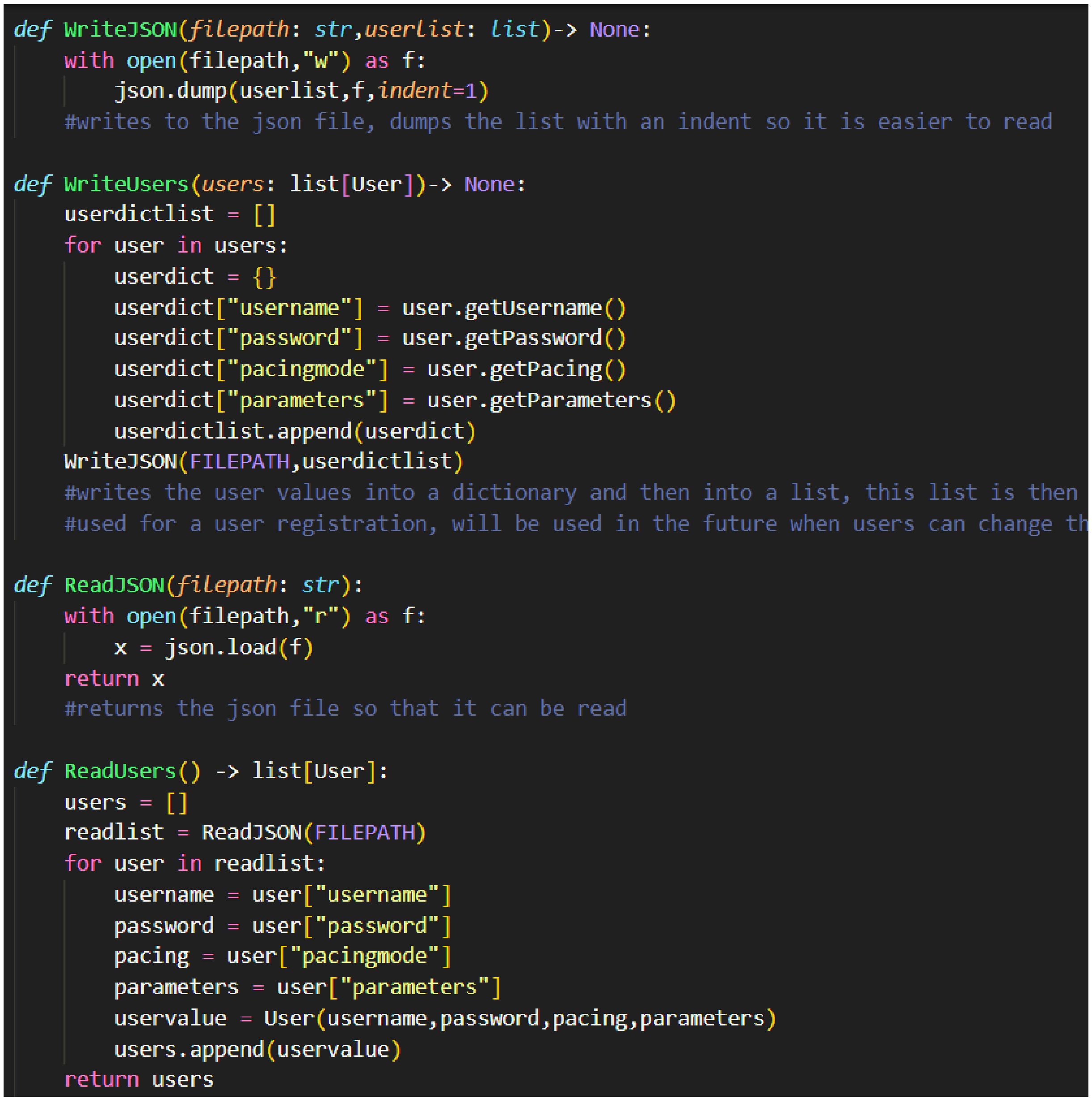
Figure 51: User Processing Functions

2.2.2.6 User_Storage Module

Module for everything relating to the json file. These are the functions that are used to write and read the values in the json file. The ReadUsers() is called whenever the values stored in the json file need to be accessed and the WriteUser() function is called whenever the values in the json file need to modified.

Function	Purpose	Parameter
WriteJSON()	Used to dump the written data into the json file	Filepath, userlist
WriteUser()	Writes the entire user to the json file, has a for loop to get each user dictionary value and then saves it all to a list which is then inputted into the WriteJSON()	User[]
ReadJSON()	Reads from the json file and returns the entire read list	filepath
readUsers()	Calls the ReadJSON() function and then uses a for loop to append all of the users and their values to a list that is then returned	

Table 22: User Storage Module Functions



```python
def WriteJSON(filepath: str,userlist: list)-> None:
    with open(filepath,"w") as f:
        json.dump(userlist,f,indent=1)
    #writes to the json file, dumps the list with an indent so it is easier to read

def WriteUsers(users: list[User])-> None:
    userdictlist = []
    for user in users:
        userdict = {}
        userdict["username"] = user.getUsername()
        userdict["password"] = user.getPassword()
        userdict["pacingmode"] = user.getPacing()
        userdict["parameters"] = user.getParameters()
        userdictlist.append(userdict)
    WriteJSON(FILEPATH,userdictlist)
    #writes the user values into a dictionary and then into a list, this list is then
    #used for a user registration, will be used in the future when users can change th

def ReadJSON(filepath: str):
    with open(filepath,"r") as f:
        x = json.load(f)
    return x
    #returns the json file so that it can be read

def ReadUsers() -> list[User]:
    users = []
    readlist = ReadJSON(FILEPATH)
    for user in readlist:
        username = user["username"]
        password = user["password"]
        pacing = user["pacingmode"]
        parameters = user["parameters"]
        uservalue = User(username,password,pacing,parameters)
        users.append(uservalue)
    return users
```

Figure 52. User Storage Module

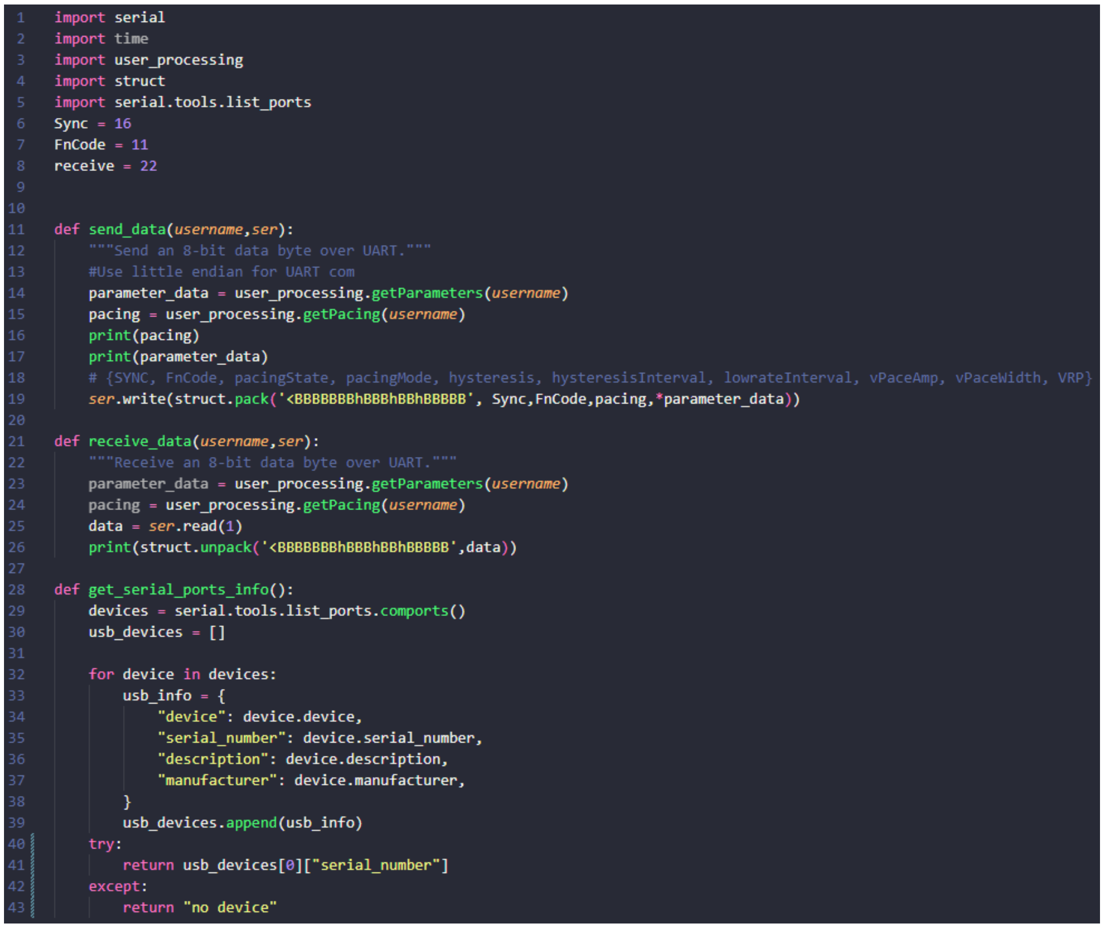
2.2.2.7 UART Module

This module is used to transmit all the user's data to the Pacemaker. It also reads the Serial ports to see if a Pacemaker is connected to the DCM.

Function	**Purpose**	**Parameter**
Send_data()	Pulls all the data for specific user and places it all in a struct pack, this struct pack turns the data into a string which is then packed into bytes and written to the UART.	Username, serial

Receive_data()	Receive data first writes data to the Pacemaker so that the specific code to enable sending back data to the DCM is sent. Then it reads all the data from the UART and unpacks it so that it is no longer in byte string format.	Username, serial
get_serial_ports_info()	Using the serial.tools.list_ports library, this function returns the list of usb devices connected to the DCM. It then returns the serial number of the connected Pacemaker so that it can be displayed on the screen.	

Table 23: UART Module Functions



```python
import serial
import time
import user_processing
import struct
import serial.tools.list_ports
Sync = 16
FnCode = 11
receive = 22


def send_data(username,ser):
    """Send an 8-bit data byte over UART."""
    #Use little endian for UART com
    parameter_data = user_processing.getParameters(username)
    pacing = user_processing.getPacing(username)
    print(pacing)
    print(parameter_data)
    # {SYNC, FnCode, pacingState, pacingMode, hysteresis, hysteresisInterval, lowrateInterval, vPaceAmp, vPaceWidth, VRP}
    ser.write(struct.pack('<BBBBBBBhBBBhBBhBBBBB', Sync,FnCode,pacing,*parameter_data))

def receive_data(username,ser):
    """Receive an 8-bit data byte over UART."""
    parameter_data = user_processing.getParameters(username)
    pacing = user_processing.getPacing(username)
    data = ser.read(1)
    print(struct.unpack('<BBBBBBBhBBBhBBhBBBBB',data))

def get_serial_ports_info():
    devices = serial.tools.list_ports.comports()
    usb_devices = []

    for device in devices:
        usb_info = {
            "device": device.device,
            "serial_number": device.serial_number,
            "description": device.description,
            "manufacturer": device.manufacturer,
        }
        usb_devices.append(usb_info)
    try:
        return usb_devices[0]["serial_number"]
    except:
        return "no device"
```

Figure 53: UART Module

2.3 Module Internal Design

Describes the state variables, private functions, and internal behaviour of all functions.

2.3.1 Simulink

2.3.1.1 Inputs and Outputs Subsystem

The Inputs and Outputs Subsystem takes inputs from the Simulink stateflow and assigns them to pin outs on the pacemaker board. It also takes signals from the pacemaker board and outputs them to the Simulink stateflow. Finally, it outputs ACTIVITY_LEVEL based on the read accelerometer data. To test this stateflow, the pin RED_LED was set to on to see if the subsystem was successfully outputting to the board. The LED turned on, indicating that the subsystem was outputting information to the board. Further tests were done through utilising the outputs from the stateflow in the main stateflow. This was functional and compiled, indicating the subsystem was taking in information from the board and outputting it to the stateflow.

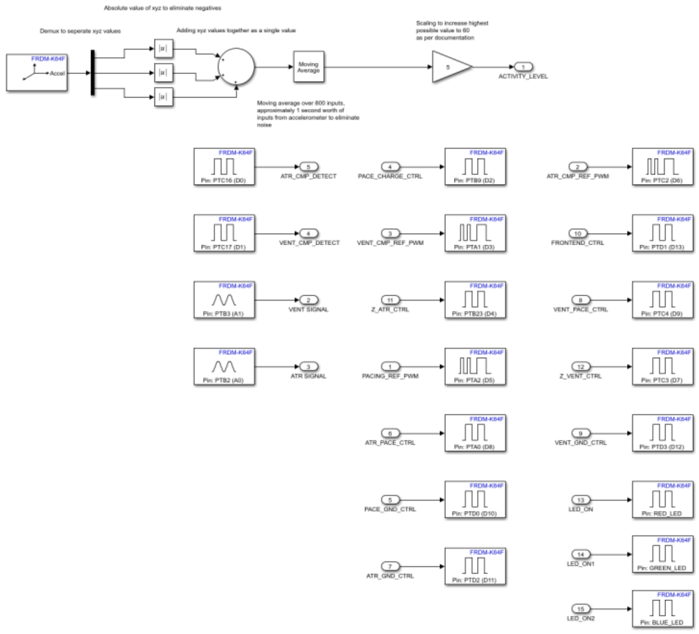


Figure 54: Subsystem Pin Assignments

2.3.1.2 Main stateflow

The user inputs the values of the constants such as the values for Atrial and Ventricle Amplitude, Lower_Rate_Limit, Upper_Rate_Limit, Atrial and Ventricle Pusle Width, Hysteresis_Interval, VRP ARP, VENT_CMP_DETECT and ATR_CMP_DETECT. The mode of the pacemaker is chosen by inputting the value in the State_Chosen variable, which is transmitted through UART from the DCM, the available states are as follows: AOO = 1, VOO = 2, AAI = 3 and VVI = 4, AOOR = 5, VOOR = 6, AAIR

= 7 and VVIR = 8. In this stateflow you can also see the connection between the pacemaker and the subsystems around making everything more organized and easier to read.

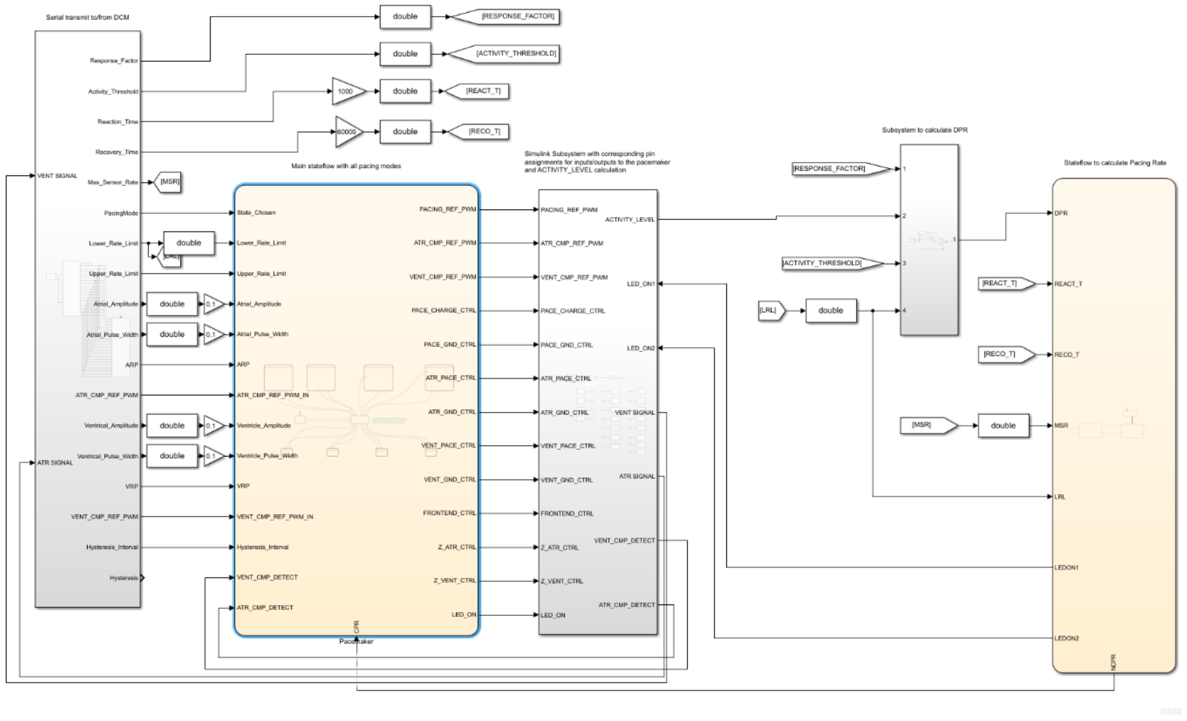


Figure 55: Main Pacemaker State Flow with User Inputs and Outputs to the Subsystem

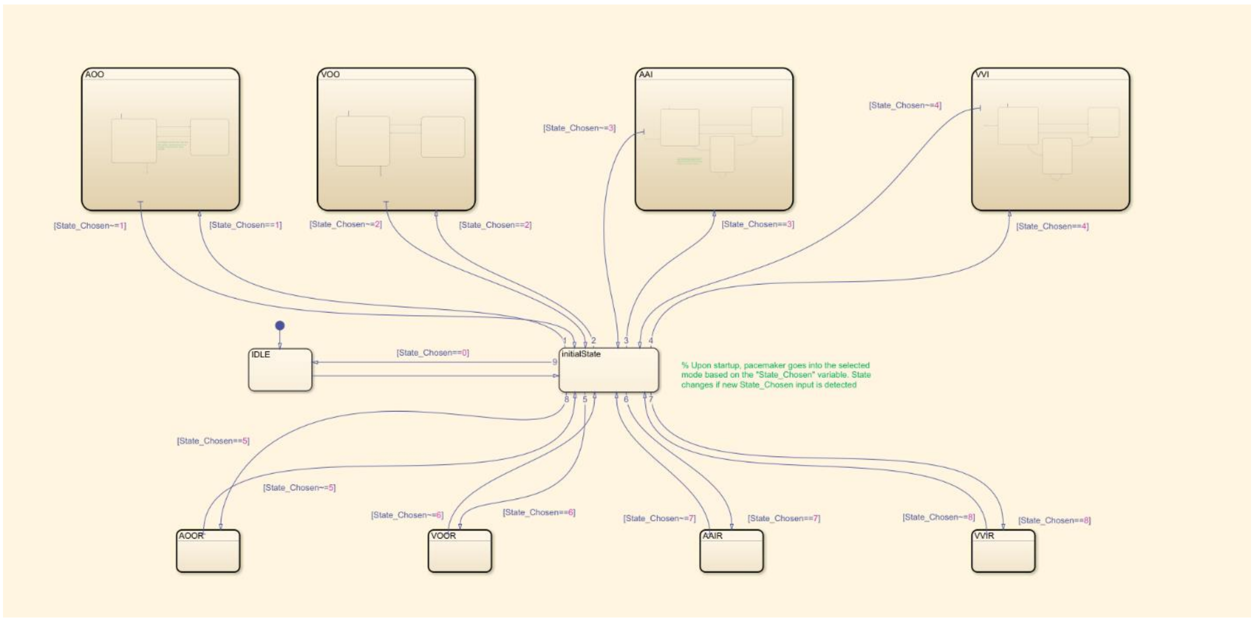
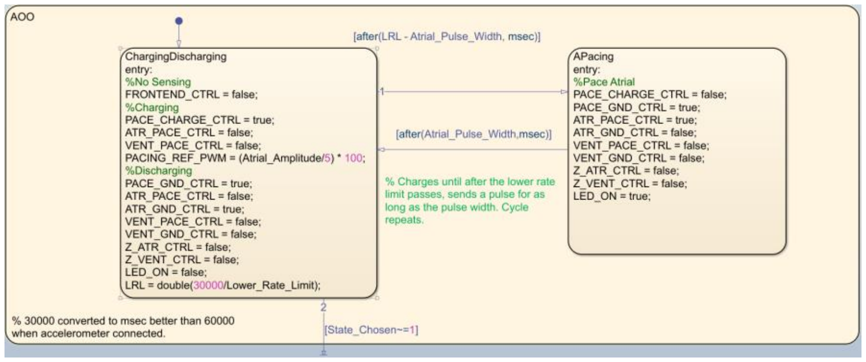


Figure 56: Secondary State Flow for Selecting the Desired Mode

2.3.1.3 AOO stateflow

The Pacemaker paces the atrium at a fixed rate without sensing or reacting to the heart's natural activity. It starts from the ChargingDischarging state, where all the output variables are assigned to charge capacitor C22 and discharge capacitor C21. "PACE_CHARGE_CTRL = true" shows that the pacing circuit is enabled and "FRONTEND_CTRL = false" means that the pacemaker does not sense or act upon natural atrial activity. It will wait in the ChargingDischarging state for (Lower_Rate_Limit – Atrial_Pulse_Width) milliseconds, then transition to the APacing state where C22 capacitor is discharged into C21 to pace the atrium. "PACE_GND_CTRL = true" shows the discharging of the C22 capacitor and "ATR_PACE_CTRL = true" means that it is pacing the atrium. It will pace the atrium until the C22 capacitor is fully discharged after (Atrial_Pulse_width) milliseconds and then transition to the charging and discharging state again.


AOO
[after(LRL - Atrial_Pulse_Width, msec)]
1
[after(Atrial_Pulse_Width,msec)]
% Charges until after the lower rate limit passes, sends a pulse for as long as the pulse width. Cycle repeats.
2
[State_Chosen~=1]
% 30000 converted to msec better than 60000 when accelerometer connected.

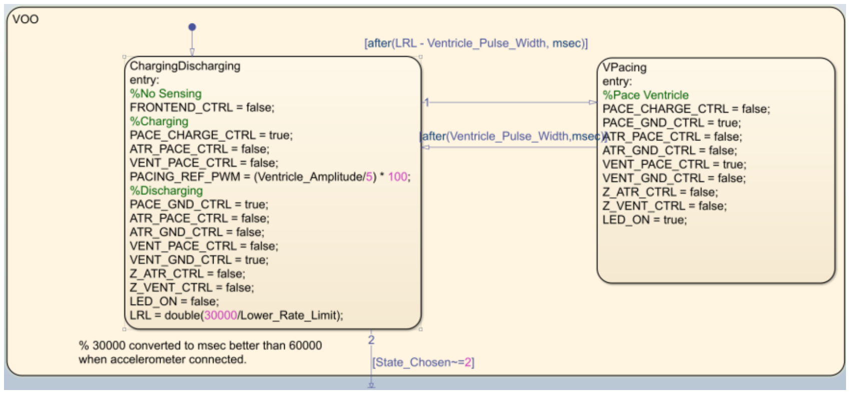
```
ChargingDischarging
entry:
%No Sensing
FRONTEND_CTRL = false;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Atrial_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = true;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
LRL = double(30000/Lower_Rate_Limit);
```

```
APacing
entry:
%Pace Atrial
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = true;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

Figure 57: AOO State Flow

2.3.1.4 VOO stateflow

The Pacemaker paces the ventricle at a fixed rate without sensing or reacting to the heart's natural activity. It starts from the ChargingDischarging state, where all the output variables are assigned to charge capacitor C22 and discharge capacitor C21. PACE_CHARGE_CTRL = true shows that the pacing circuit is enabled and FRONTEND_CTRL = false means that the pacemaker does not sense or act upon natural atrial activity. It will wait in the ChargingDischarging state for (Lower_Rate_Limit – Ventricle_Pulse_Width) milliseconds, then transition to the VPacing state where C22 capacitor is discharged into C21 to pace the atrium. PACE_GND_CTRL = true shows the discharging of the C22 capacitor and VENT_PACE_CTRL = true means that it is pacing the ventricle. It will pace the ventricle until the C22 capacitor is fully discharged after (Ventricle_Pulse_width) milliseconds and then transition to the charging and discharging state again.


VOO
[after(LRL - Ventricle_Pulse_Width, msec)]
1
[after(Ventricle_Pulse_Width,msec)]
2
[State_Chosen~=2]

```
ChargingDischarging
entry:
%No Sensing
FRONTEND_CTRL = false;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Ventricle_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = true;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
LRL = double(30000/Lower_Rate_Limit);
```

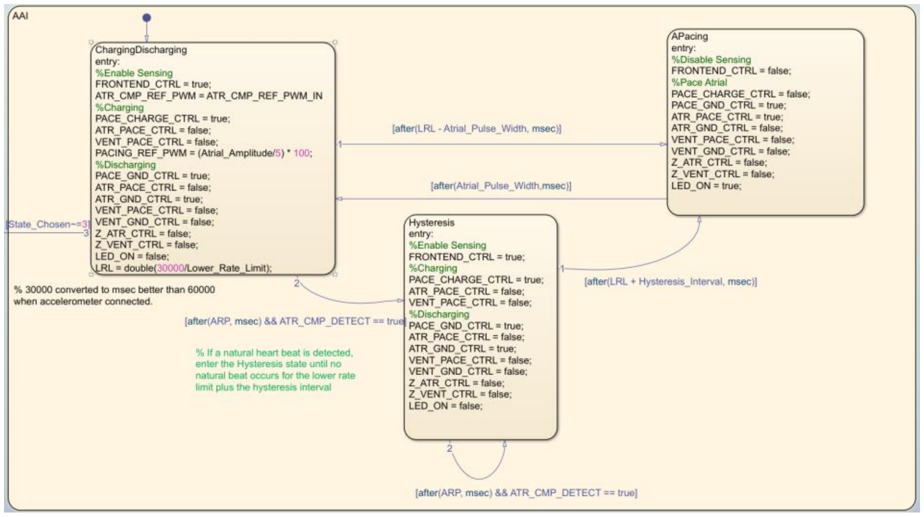
```
VPacing
entry:
%Pace Ventricle
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = true;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

```
% 30000 converted to msec better than 60000
when accelerometer connected.
```

Figure 58: VOO State Flow

2.3.1.5 AAI stateflow

The pacemaker is initially in the charge/discharging state where it initializes/sets the variables. In this state, PACING_REF_PWM is set to ($Atrial\_Amplitude/5 * 100$) to set the duty cycle % with the max amplitude being 5V, and atrial sensing is enabled through setting FRONT_END_CTRL to true to sense the heartbeat. PACE_CHARGE_CTRL is also set to true to enable charging. If no heartbeat/atrial activity is detected, hence ATR_CMP_DETECT is false, then after LRL – Atrial Pulse Width, the pacing state changes to the APacing state. In this state, sensing is now disabled by setting FRONTEND_CTRL to false. PACE_CHARGE_CTRL is now set to false, and ATR_PACE_CTRL is set to true to indicate that is now in the atrial pacing state. After another atrial pulse width, it gets sent back to the charge/discharge state. The pacing state cycles between these 2 states charging/discharging and APacing until a non-refractory atrial event is detected (after ARP and when ATR_CMP_DETECT is true). When this is detected, the pacing state gets sent to hysteresis where sensing is still enabled. In the hysteresis state, charging/discharging is enabled as PACE_CHARGE_CTRL, ATR_GND_CTRL, and PACE_GND_CTRL are all true whereas the other variables are set to false. The pacemaker stays in the hysteresis state as long as a non-refractory heartbeat is detected (where ATR_CMP_DETECT is true). When no natural heartbeat is detected in the interval ($Lower\_rate\_limit + Hysteresis\ Interval$), then the pacing state gets sent to APacing State after that interval, where it cycles again between the charge/discharge state until a natural heartbeat is detected.


AAI
[after(LRL - Atrial_Pulse_Width, msec)]
[after(Atrial_Pulse_Width,msec)]
[State_Chosen~=3]
[after(ARP, msec) && ATR_CMP_DETECT == true]
[after(LRL + Hysteresis_Interval, msec)]
[after(ARP, msec) && ATR_CMP_DETECT == true]
1
2
3
1
2

```
ChargingDischarging
entry:
%Enable Sensing
FRONTEND_CTRL = true;
ATR_CMP_REF_PWM = ATR_CMP_REF_PWM_IN
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Atrial_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = true;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
LRL = double(30000/Lower_Rate_Limit);
```

```
% 30000 converted to msec better than 60000
when accelerometer connected.
```

```
% If a natural heart beat is detected,
enter the Hysteresis state until no
natural beat occurs for the lower rate
limit plus the hysteresis interval
```

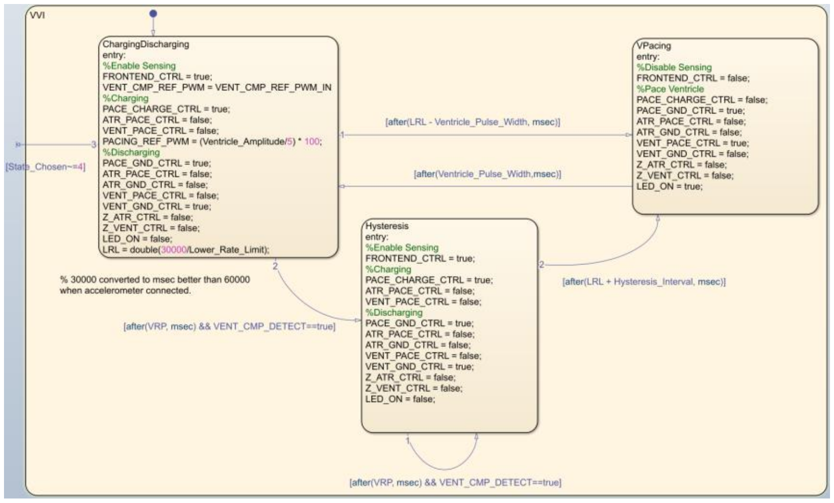
```
APacing
entry:
%Disable Sensing
FRONTEND_CTRL = false;
%Pace Atrial
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = true;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

```
Hysteresis
entry:
%Enable Sensing
FRONTEND_CTRL = true;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = true;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
```

Figure 59: AAI State Flow

2.3.1.6 VVI stateflow

The pacemaker is initially in the charge/discharging state where it initializes/sets the variables. In this state, PACING_REF_PWM is set to ($Ventricle\_Amplitude/5\ *100$) to set the duty cycle % with the max amplitude being 5V, and ventricle sensing is enabled through setting FRONT_END_CTRL to true to sense the heartbeat. If no heartbeat/ventricle activity is detected (when VENT_CMP_DETECT is false), the pacing state changes to the VPacing state after then after Lower_Rate_Limit – Ventrical Pulse Width. In this state, sensing is now disabled by setting FRONTEND_CTRL to false. PACE_CHARGE_CTRL is now set to false, and VENT_PACE_CTRL is set to true to indicate that is now in the ventricle pacing state. After another ventricle pulse width, it gets sent back to the charge/discharge state. The pacing state cycles between these 2 states charging/discharging and VPacing until a non-refractory ventricle event is detected (after VRP and when VENT_CMP_DETECT is true). When this is detected, the pacing state gets sent to hysteresis where sensing is still enabled. In the hysteresis state, charging/discharging is enabled as PACE_CHARGE_CTRL, VENT_GND_CTRL, and PACE_GND_CTRL are all true whereas the other variables are set to false. The pacemaker stays in the hysteresis state as long as a non-refractory heartbeat is detected (where VENT_CMP_DETECT is true). When no natural heartbeat is detected in the interval ($Lower\_rate\_limit\ +\ Hysteresis\ Interval$), then the pacing state gets sent to VPacing State after that interval, where it cycles again between the charge/discharge state until a natural heartbeat is detected.


VVI
[State_Chosen~=4]
[after(LRL - Ventricle_Pulse_Width, msec)]
[after(Ventricle_Pulse_Width,msec)]
[after(LRL + Hysteresis_Interval, msec)]
[after(VRP, msec) && VENT_CMP_DETECT==true]
[after(VRP, msec) && VENT_CMP_DETECT==true]
% 30000 converted to msec better than 60000 when accelerometer connected.

```
ChargingDischarging
entry:
%Enable Sensing
FRONTEND_CTRL = true;
VENT_CMP_REF_PWM = VENT_CMP_REF_PWM_IN
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Ventricle_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = true;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
LRL = double(30000/Lower_Rate_Limit);
```

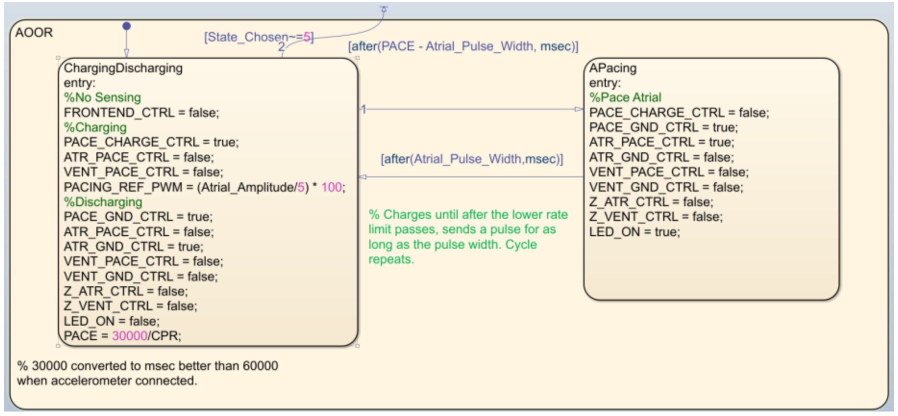
```
VPacing
entry:
%Disable Sensing
FRONTEND_CTRL = false;
%Pace Ventricle
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = true;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

```
Hysteresis
entry:
%Enable Sensing
FRONTEND_CTRL = true;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = true;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
```

Figure 60: VVI State Flow

2.3.1.7 AOOR Stateflow

The Pacemaker paces the atrium at a fixed rate without sensing or reacting to the heart's natural activity. It starts from the ChargingDischarging state, where all the output variables are assigned to charge capacitor C22 and discharge capacitor C21. "PACE_CHARGE_CTRL = true" shows that the pacing circuit is enabled and "FRONTEND_CTRL = false" means that the pacemaker does not sense or act upon natural atrial activity. It will wait in the ChargingDischarging state for (PACE – Atrial_Pulse_Width) milliseconds, then transition to the APacing state where C22 capacitor is discharged into C21 to pace the atrium. "PACE_GND_CTRL = true" shows the discharging of the C22 capacitor and "ATR_PACE_CTRL = true" means that it is pacing the atrium. It will pace the atrium until the C22 capacitor is fully discharged after (Atrial_Pulse_width) milliseconds and then transition to the charging and discharging state again.


AOOR
[State_Chosen~=5]
[after(PACE - Atrial_Pulse_Width, msec)]
[after(Atrial_Pulse_Width,msec)]
% Charges until after the lower rate limit passes, sends a pulse for as long as the pulse width. Cycle repeats.
% 30000 converted to msec better than 60000 when accelerometer connected.

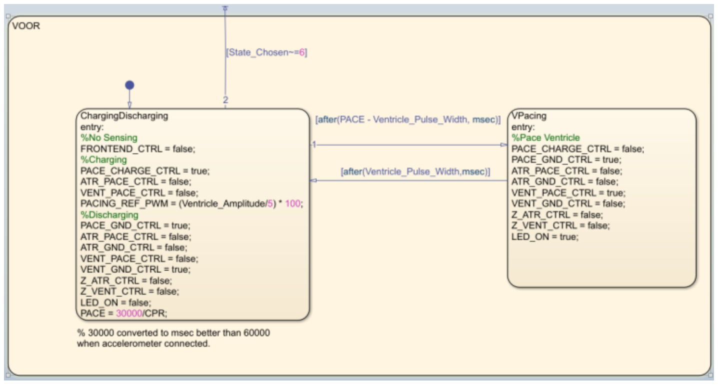
```
ChargingDischarging
entry:
%No Sensing
FRONTEND_CTRL = false;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Atrial_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = true;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
PACE = 30000/CPR;
```

```
APacing
entry:
%Pace Atrial
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = true;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

Figure 61: AOOR State Flow

2.3.1.8 VOOR Stateflow

The Pacemaker paces the ventricle at a fixed rate without sensing or reacting to the heart's natural activity. It starts from the ChargingDischarging state, where all the output variables are assigned to charge capacitor C22 and discharge capacitor C21. PACE_CHARGE_CTRL = true shows that the pacing circuit is enabled and FRONTEND_CTRL = false means that the pacemaker does not sense or act upon natural atrial activity. It will wait in the ChargingDischarging state for (PACE – Ventricle_Pulse_Width) milliseconds, then transition to the VPacing state where C22 capacitor is discharged into C21 to pace the atrium. PACE_GND_CTRL = true shows the discharging of the C22 capacitor and VENT_PACE_CTRL = true means that it is pacing the ventricle. It will pace the ventricle until the C22 capacitor is fully discharged after (Ventricle_Pulse_width) milliseconds and then transition to the charging and discharging state again.


VOOR
[State_Chosen~=6]
2
1
[after(PACE - Ventricle_Pulse_Width, msec)]
[after(Ventricle_Pulse_Width,msec)]

```
ChargingDischarging
entry:
%No Sensing
FRONTEND_CTRL = false;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Ventricle_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = true;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
PACE = 30000/CPR;
```

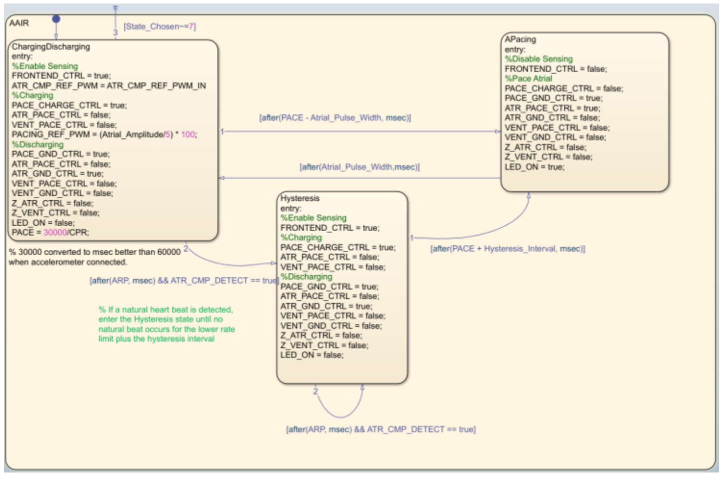
```
% 30000 converted to msec better than 60000
when accelerometer connected.
```

```
VPacing
entry:
%Pace Ventricle
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = true;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

Figure 62: VOOR State Flow

2.3.1.9 AAIR Stateflow

The pacemaker is initially in the charge/discharging state where it initializes/sets the variables. In this state, PACING_REF_PWM is set to ($Atrial\_Amplitude/5 * 100$) to set the duty cycle % with the max amplitude being 5V, and atrial sensing is enabled through setting FRONT_END_CTRL to true to sense the heartbeat. PACE_CHARGE_CTRL is also set to true to enable charging. If no heartbeat/atrial activity is detected, hence ATR_CMP_DETECT is false, then after PACE – Atrial Pulse Width, the pacing state changes to the APacing state. In this state, sensing is now disabled by setting FRONTEND_CTRL to false. PACE_CHARGE_CTRL is now set to false, and ATR_PACE_CTRL is set to true to indicate that is now in the atrial pacing state. After another atrial pulse width, it gets sent back to the charge/discharge state. The pacing state cycles between these 2 states charging/discharging and APacing until a non-refractory atrial event is detected (after ARP and when ATR_CMP_DETECT is true). When this is detected, the pacing state gets sent to hysteresis where sensing is still enabled. In the hysteresis state, charging/discharging is enabled as PACE_CHARGE_CTRL, ATR_GND_CTRL, and PACE_GND_CTRL are all true whereas the other variables are set to false. The pacemaker stays in the hysteresis state as long as a non-refractory heartbeat is detected (where ATR_CMP_DETECT is true). When no natural heartbeat is detected in the interval ($PACE + Hysteresis\ Interval$), then the pacing state gets sent to APacing State after that interval, where it cycles again between the charge/discharge state until a natural heartbeat is detected.


AAIR
[State_Chosen~=7]
3
[after(PACE - Atrial_Pulse_Width, msec)]
1
[after(Atrial_Pulse_Width,msec)]
2
[after(ARP, msec) && ATR_CMP_DETECT == true]
1
[after(PACE + Hysteresis_Interval, msec)]
2
[after(ARP, msec) && ATR_CMP_DETECT == true]
% If a natural heart beat is detected, enter the Hysteresis state until no natural beat occurs for the lower rate limit plus the hysteresis interval

```
ChargingDischarging
entry:
%Enable Sensing
FRONTEND_CTRL = true;
ATR_CMP_REF_PWM = ATR_CMP_REF_PWM_IN
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Atrial_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = true;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
PACE = 30000/CPR;
```

```
% 30000 converted to msec better than 60000
when accelerometer connected.
```

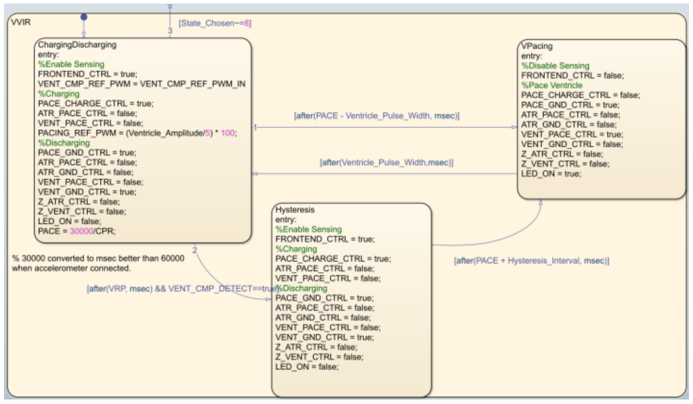
```
APacing
entry:
%Disable Sensing
FRONTEND_CTRL = false;
%Pace Atrial
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = true;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

```
Hysteresis
entry:
%Enable Sensing
FRONTEND_CTRL = true;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = true;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
```

Figure 63: AAIR State Flow

2.3.1.10 VVIR Stateflow

The pacemaker is initially in the charge/discharging state where it initializes/sets the variables. In this state, PACING_REF_PWM is set to ($Ventricle\_Amplitude/5\ *\ 100$) to set the duty cycle % with the max amplitude being 5V, and ventricle sensing is enabled through setting FRONT_END_CTRL to true to sense the heartbeat. If no heartbeat/ventricle activity is detected (when VENT_CMP_DETECT is false), the pacing state changes to the VPacing state after then after PACE – Ventrical Pulse Width. In this state, sensing is now disabled by setting FRONTEND_CTRL to false. PACE_CHARGE_CTRL is now set to false, and VENT_PACE_CTRL is set to true to indicate that is now in the ventricle pacing state. After another ventricle pulse width, it gets sent back to the charge/discharge state. The pacing state cycles between these 2 states charging/discharging and VPacing until a non-refractory ventricle event is detected (after VRP and when VENT_CMP_DETECT is true). When this is detected, the pacing state gets sent to hysteresis where sensing is still enabled. In the hysteresis state, charging/discharging is enabled as PACE_CHARGE_CTRL, VENT_GND_CTRL, and PACE_GND_CTRL are all true whereas the other variables are set to false. The pacemaker stays in the hysteresis state as long as a non-refractory heartbeat is detected (where VENT_CMP_DETECT is true). When no natural heartbeat is detected in the interval ($PACE\ +\ Hysteresis\ Interval$), then the pacing state gets sent to VPacing State after that interval, where it cycles again between the charge/discharge state until a natural heartbeat is detected.


VVIR
[State_Chosen~=8]
3
[after(PACE - Ventricle_Pulse_Width, msec)]
1
[after(Ventricle_Pulse_Width,msec)]
2
[after(VRP, msec) && VENT_CMP_DETECT==true]
[after(PACE + Hysteresis_Interval, msec)]

```
ChargingDischarging
entry:
%Enable Sensing
FRONTEND_CTRL = true;
VENT_CMP_REF_PWM = VENT_CMP_REF_PWM_IN
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
PACING_REF_PWM = (Ventricle_Amplitude/5) * 100;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = true;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
PACE = 30000/CPR;
```

```
% 30000 converted to msec better than 60000
when accelerometer connected.
```

```
VPacing
entry:
%Disable Sensing
FRONTEND_CTRL = false;
%Pace Ventricle
PACE_CHARGE_CTRL = false;
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = true;
VENT_GND_CTRL = false;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = true;
```

```
Hysteresis
entry:
%Enable Sensing
FRONTEND_CTRL = true;
%Charging
PACE_CHARGE_CTRL = true;
ATR_PACE_CTRL = false;
VENT_PACE_CTRL = false;
%Discharging
PACE_GND_CTRL = true;
ATR_PACE_CTRL = false;
ATR_GND_CTRL = false;
VENT_PACE_CTRL = false;
VENT_GND_CTRL = true;
Z_ATR_CTRL = false;
Z_VENT_CTRL = false;
LED_ON = false;
```

Figure 64: VVIR State Flow

2.3.1.11 Serial transmit to/from DCM Subsystem

This subsystem allows communication between the pacemaker and the DCM though UART. The inputs to the subsystem are VENT_SIGNAL and ATR_SIGNAL which are received from the pacemaker board and sent to the DCM over UART when egram data is requested. rxdata holds the received data as an array where each index corresponds to a pacemaker parameter. If the parameter requires more than 8 bits such as a double data type, multiple indexes combined with typecast to recover the original value. Status represents the communication state of the system and goes low when a transmission is received from the DCM. The stateflow has outputs which connect to the main pacemaker and to the rate adaptive stateflows to send the required parameters. These parameters can also be sent back to the DCM when requested for verification.

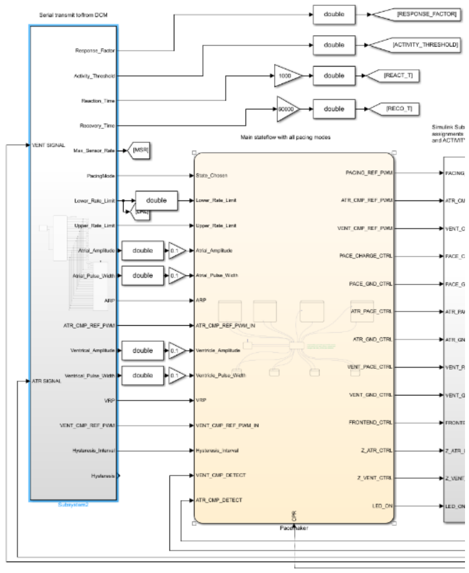


Figure 65: UART Subsystem

2.3.1.12 Receiving Data from DCM stateflow

This state flow processes the UART data received from DCM. It begins in the initial state where nominal values for the pacemaker parameters are set (PacingMode, Lower and Upper rate limit, Atrial and Ventricle pulse width…). The initial pacemaker mode is set to idle for safety since the pacemaker should only start pacing once the DCM is connected, and the correct parameters are set. The state flow processes the requests from the DCM based on the "Fn_Code" with options to set parameters, send parameters, and send egram data. If no valid FnCode is detected it transitions to the STANDBY state to avoid miscommunication errors. SET_PARAM updates the pacemakers parameters using values from rxdata. ECHO_PARAM sends back the current parameters for confirmation. SEND_EGRAM continuously transmits electrogram data until it is interrupted by another command.

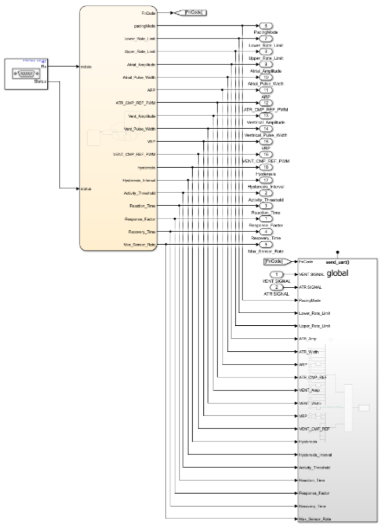
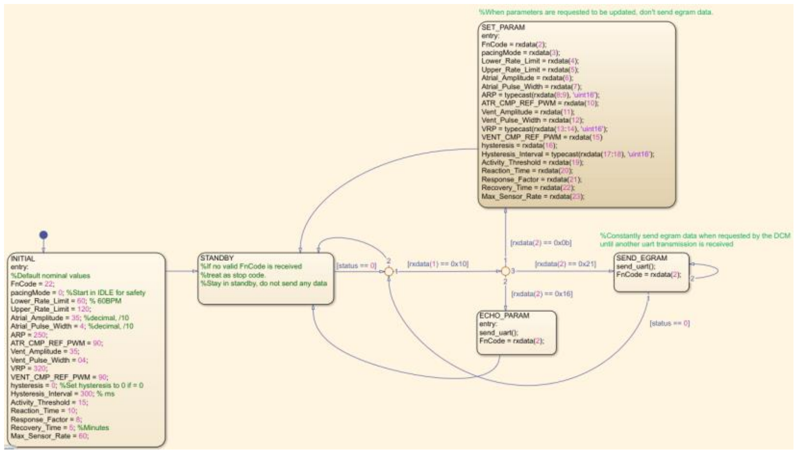


Figure 66: UART subsystem with receiving state flow and transmit function



```
%When parameters are requested to be updated, don't send egram data.
SET_PARAM
entry:
FnCode = rxdata(2);
pacingMode = rxdata(3);
Lower_Rate_Limit = rxdata(4);
Upper_Rate_Limit = rxdata(5);
Atrial_Amplitude = rxdata(6);
Atrial_Pulse_Width = rxdata(7);
ARP = typecast(rxdata(8:9), 'uint16');
ATR_CMP_REF_PWM = rxdata(10);
Vent_Amplitude = rxdata(11);
Vent_Pulse_Width = rxdata(12);
VRP = typecast(rxdata(13:14), 'uint16');
VENT_CMP_REF_PWM = rxdata(15)
hysteresis = rxdata(16);
Hysteresis_Interval = typecast(rxdata(17:18), 'uint16');
Activity_Threshold = rxdata(19);
Reaction_Time = rxdata(20);
Response_Factor = rxdata(21);
Recovery_Time = rxdata(22);
Max_Sensor_Rate = rxdata(23);
```

```
INITIAL
entry:
%Default nominal values
FnCode = 22;
pacingMode = 0; %Start in IDLE for safety
Lower_Rate_Limit = 60; % 60BPM
Upper_Rate_Limit = 120;
Atrial_Amplitude = 35; %decimal, /10
Atrial_Pulse_Width = 4; %decimal, /10
ARP = 250;
ATR_CMP_REF_PWM = 90;
Vent_Amplitude = 35;
Vent_Pulse_Width = 04;
VRP = 320;
VENT_CMP_REF_PWM = 90;
hysteresis = 0; %Set hysteresis to 0 if = 0
Hysteresis_Interval = 300; % ms
Activity_Threshold = 15;
Reaction_Time = 10;
Response_Factor = 8;
Recovery_Time = 5; %Minutes
Max_Sensor_Rate = 60;
```

```
STANDBY
%If no valid FnCode is received
%treat as stop code.
%Stay in standby, do not send any data
```

```
ECHO_PARAM
entry:
send_uart();
FnCode = rxdata(2);
```

```
%Constantly send egram data when requested by the DCM
until another uart transmission is received
SEND_EGRAM
send_uart();
FnCode = rxdata(2);
```

Figure 67: UART receive state flow

2.3.1.13 Transmitting Data to DCM through UART subsystem

This subsystem is triggered from the send_uart() function call back in the receive UART state flow. A switch is used to swap between transmitting pacemaker parameters and sending egram data. The "FnCode" is used to determine which set of data to transmit with "22" indicating to send parameters, and "33" indicating to send egram data. Both input arrays must be the same size, requiring the egram data to be zero padded. In addition, for data that are larger than 8 bits, byte pack is required to send them in the correct format which can be reconstructed by the DCM.

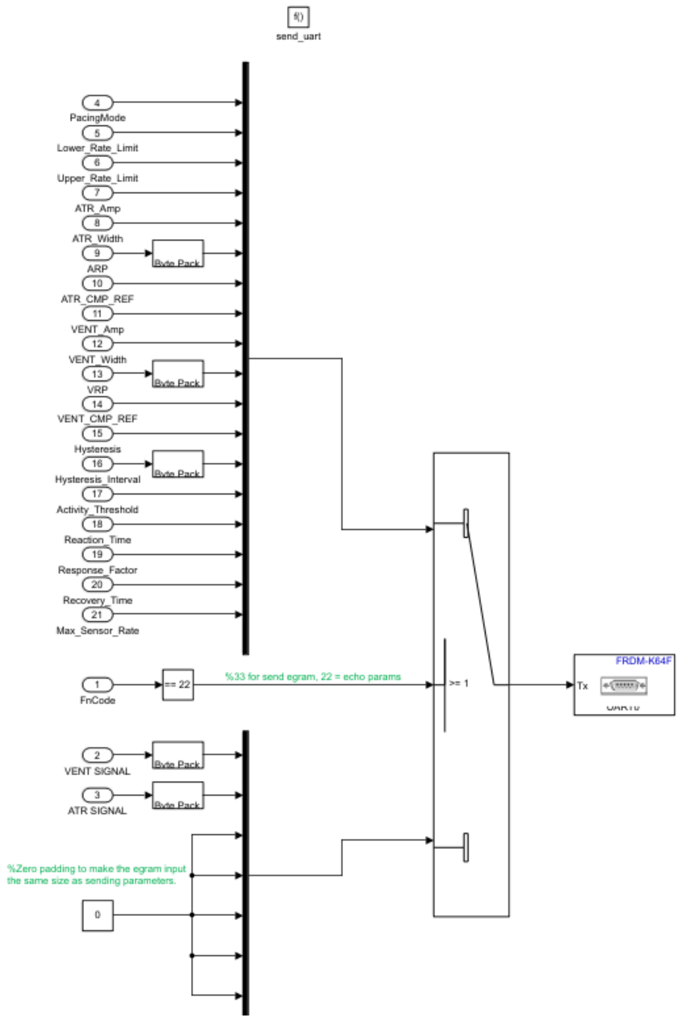


Figure 68: Send parameters/egram triggered subsystem

2.3.1.14 DPR Subsystem

This subsystem calculates the Desired Pacing Rate (DPR). If AL > AT, the formula for DPR is $43 * Response\_Factor * log(Activity\_Level - Activity\_Threshold) + Lower\_Rate\_Limit$. If AL <= AT, DPR is set to Lower_Rate_Limit. DPR is then output to be used in the Pacing Rate Subsystem.

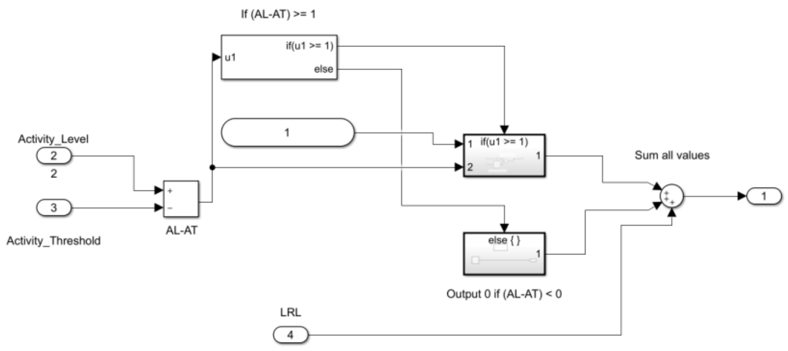


Figure 69: DPR Subsystem Overview

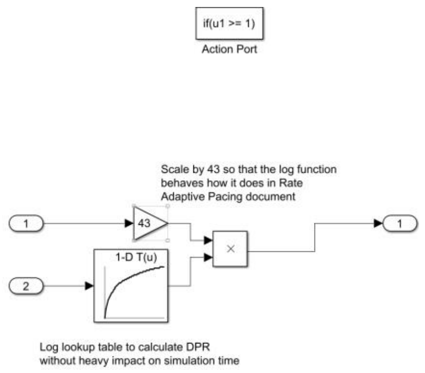
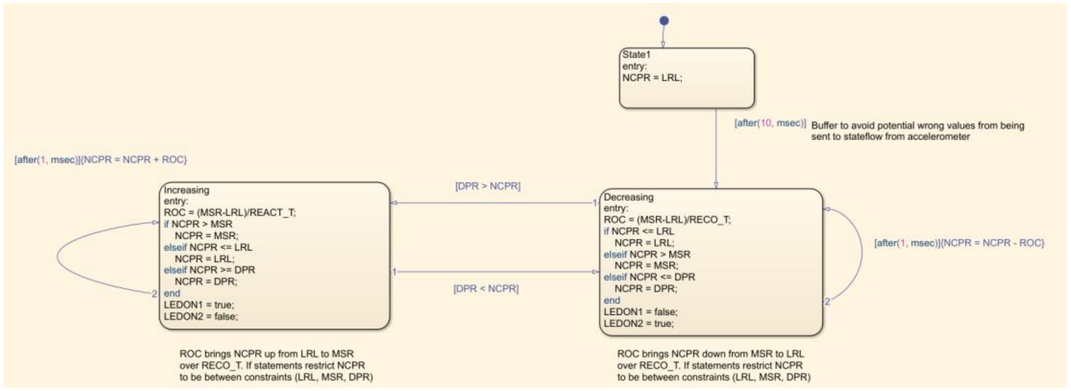

if(u1 >= 1)
Action Port
Scale by 43 so that the log function behaves how it does in Rate Adaptive Pacing document
1
43
×
1
1-D T(u)
2
Log lookup table to calculate DPR without heavy impact on simulation time

Figure 70: DPR Subsystem - if AL > AT

2.3.1.15 Pacing Rate Stateflow

This stateflow increases and decreases the Current_Pacing_Rate (CPR and NCPR). It starts in the entry state where it sets NCPR to LRL bpm. After 10 seconds, to avoid an immediate accelerometer misfire, it enters the decreasing state. In this state, Rate of Change (ROC) gets calculated as (MSR – LRL)/RECO_T. This calculates a slope in which if the CPR was at MSR, it would reach LRL in RECO_T seconds. The state also checks upon entry if NCPR <= LRL, if NCPR > MSR, or if NCPR <= DPR. If these are true individually, then NCPR will be set to LRL, MSR, and DPR respectively. The LED also gets set to blue to indicate the board is in Decreasing mode. After 1 millisecond, the NCPR gets decreased by ROC. This continues until DPR is greater than NCPR, at which point the stateflow enters the Increasing state. In this state, Rate of Change (ROC) gets calculated as (MSR – LRL)/REACT_T. This calculates a slope in which if the CPR was at LRL, it would reach MSR in REACT_T seconds. The state also checks upon entry if NCPR > MSR, if NCPR <= LRL, or if NCPR >= DPR. If these are true individually, then NCPR will be set to MSR, LRL, and DPR respectively. The LED also gets set to green to indicate the board is in Increasing mode. After 1 millisecond, the NCPR gets increased by ROC. This continues until DPR is greater than NCPR, at which point the stateflow enters the Decreasing state, explained above.



```
State1
entry:
NCPR = LRL;
```

```
Increasing
entry:
ROC = (MSR-LRL)/REACT_T;
if NCPR > MSR
   NCPR = MSR;
elseif NCPR <= LRL
   NCPR = LRL;
elseif NCPR >= DPR
   NCPR = DPR;
end
LEDON1 = true;
LEDON2 = false;
```

```
Decreasing
entry:
ROC = (MSR-LRL)/RECO_T;
if NCPR <= LRL
   NCPR = LRL;
elseif NCPR > MSR
   NCPR = MSR;
elseif NCPR <= DPR
   NCPR = DPR;
end
LEDON1 = false;
LEDON2 = true;
```

[after(10, msec)]
Buffer to avoid potential wrong values from being sent to stateflow from accelerometer
[after(1, msec)]{NCPR = NCPR + ROC}
[DPR > NCPR]
[DPR < NCPR]
[after(1, msec)]{NCPR = NCPR - ROC}
1
2
1
1
2
ROC brings NCPR up from LRL to MSR over RECO_T. If statements restrict NCPR to be between constraints (LRL, MSR, DPR)
ROC brings NCPR down from MSR to LRL over RECO_T. If statements restrict NCPR to be between constraints (LRL, MSR, DPR)

Figure 71: Pacing Rate State Flow

2.4 Assurance Case

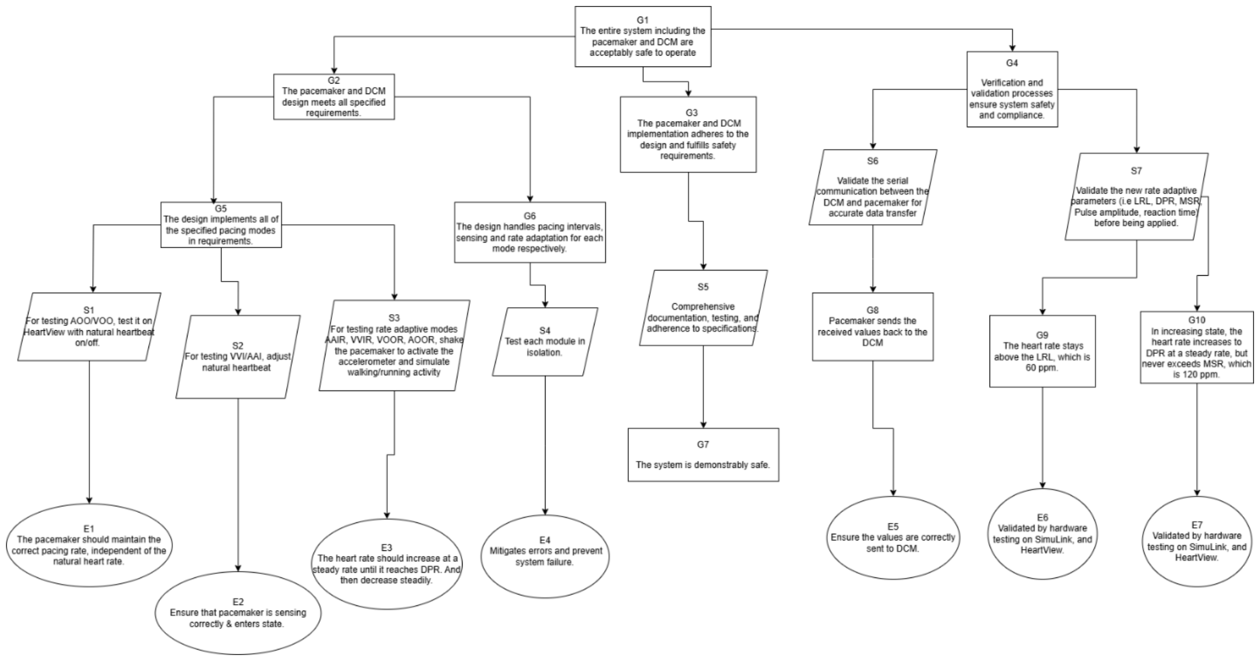

G1
The entire system including the pacemaker and DCM are acceptably safe to operate
G2
The pacemaker and DCM design meets all specified requirements.
G3
The pacemaker and DCM implementation adheres to the design and fulfills safety requirements.
G4
Verification and validation processes ensure system safety and compliance.
G5
The design implements all of the specified pacing modes in requirements.
G6
The design handles pacing intervals, sensing and rate adaptation for each mode respectively.
S6
Validate the serial communication between the DCM and pacemaker for accurate data transfer
S7
Validate the new rate adaptive parameters (i.e LRL, DPR, MSR, Pulse amplitude, reaction time) before being applied.
S1
For testing AOO/VOO, test it on HeartView with natural heartbeat on/off.
S2
For testing VVI/AAI, adjust natural heartbeat
S3
For testing rate adaptive modes AAIR, VVIR, VOOR, AOOR, shake the pacemaker to activate the accelerometer and simulate walking/running activity
S4
Test each module in isolation.
S5
Comprehensive documentation, testing, and adherence to specifications.
G8
Pacemaker sends the received values back to the DCM
G9
The heart rate stays above the LRL, which is 60 ppm.
G10
In increasing state, the heart rate increases to DPR at a steady rate, but never exceeds MSR, which is 120 ppm.
G7
The system is demonstrably safe.
E1
The pacemaker should maintain the correct pacing rate, independent of the natural heart rate.
E2
Ensure that pacemaker is sensing correctly & enters state.
E3
The heart rate should increase at a steady rate until it reaches DPR. And then decrease steadily.
E4
Mitigates errors and prevent system failure.
E5
Ensure the values are correctly sent to DCM.
E6
Validated by hardware testing on SimuLink, and HeartView.
E7
Validated by hardware testing on SimuLink, and HeartView.

Figure 72: Assurance Case Diagram