Software Development 3K04
Assignment 2

Instructor: Dr. Alan Wassyng

December 1st, 2021

Siddharth Dhanasekar - dhanass
William Siddeley - siddelew
Adarsh Patel - patea139
Mit Patel - patem70
Sagaanan Ganeshathasan - ganess19
Jeryn Anthonypillai - anthoj7

Table of Contents

Part 1 - Pacemaker Design

Current Requirements

1. Use Simulink to implement stateflows for the following pacemaker modes for a pacemaker: AOO, VOO, AAI, VVI, DOO, AOOR, VOOR, AAIR, VVIR, and DOOR.
2. The stateflow should use the programmable parameters listed in the requirements document, mainly the pulse characteristics, rate characteristics and what chamber is being paced
3. The stateflow should not change if the pinmap is altered, but rather the correct pin should map to its corresponding component
4. Include serial communications between DCM and Pacemaker, so that modes and pulse characteristics can be dynamically changed without restarting the device
5. Rate adaptive modes must track activity using the on-board accelerometer. The pulse rate (LRL) should increase once sufficient activity is detected, that is, when activity level is greater than activity threshold

Design decisions likely to change

1. From the feedback received from the lab demo, we will be creating separate states for the rate adaptive modes from the non-rate adaptive modes, that is AOO, VOO, AAI, VVI and DOO will not be AOOR, VOOR, AAIR, VVIR and DOOR states with adaptive pacing rate equalling LRL to avoid user confusion.
2. In addition, instead of just detecting motion in the z-axis for activity, we will use all three axes (xyz) to detect activity.
3. DDDR mode with the proper AV delay (with rate adaptive)
4. Inhibit only ventricular pacing when holding down the pushbutton (i.e. atrial functionality is unaffected). Once the pushbutton is released then ventricular pacing will appear

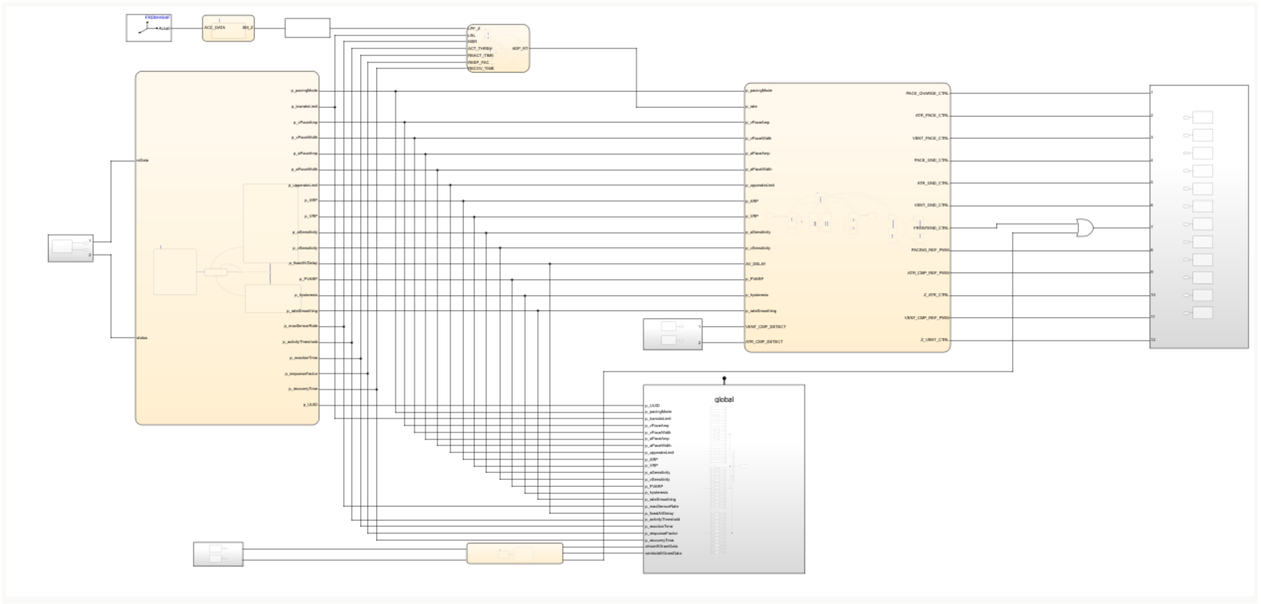


Figure 1: Simulink Model

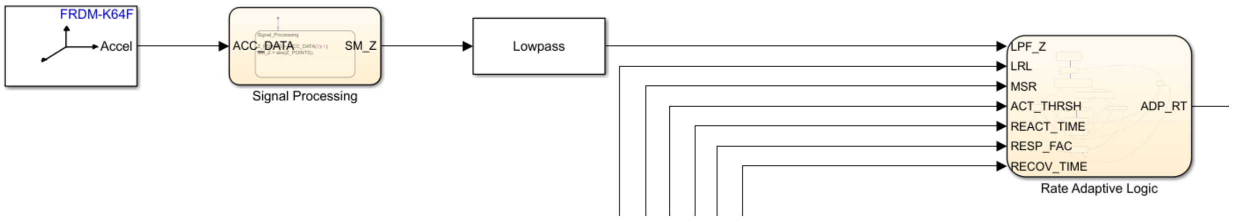


Figure 2: Rate Adaptive Block

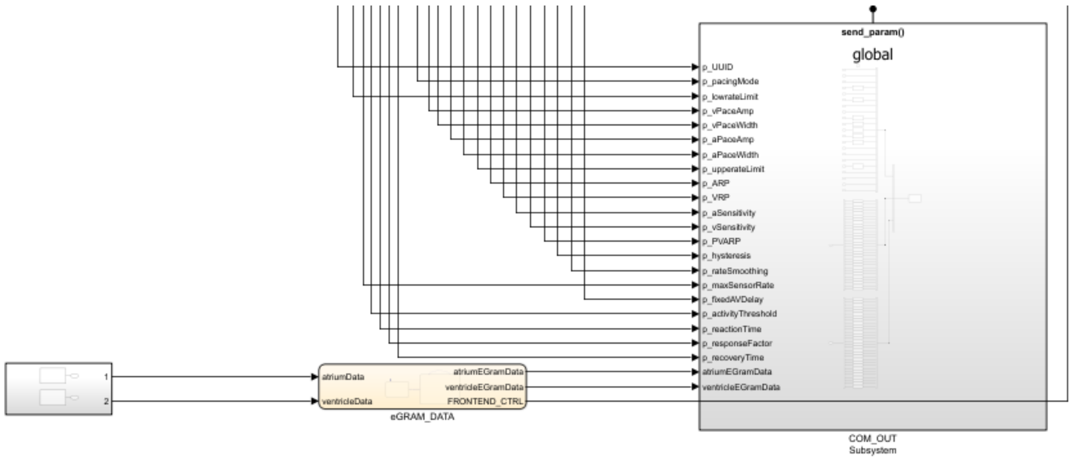


Figure 3: Serial Communication OUT Block

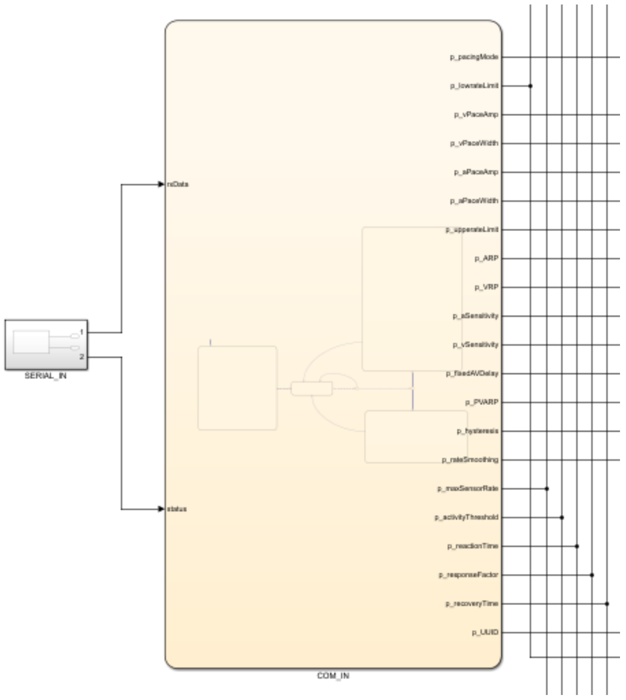


Figure 4: Serial Communication IN Block

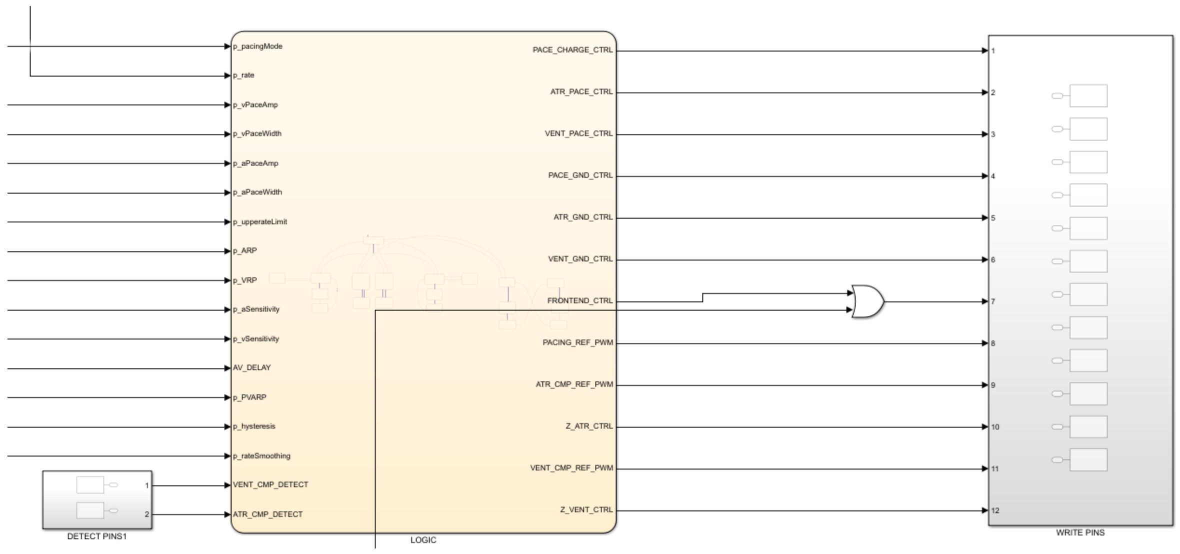


Figure 5: Pacing Mode Block

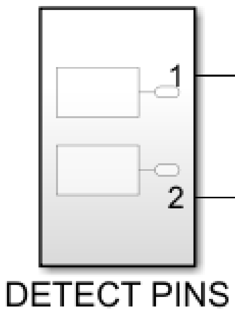
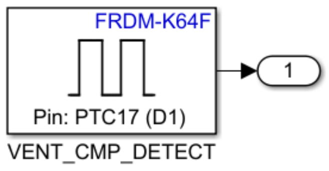
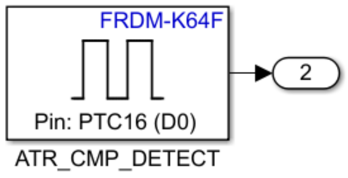
Programmable Parameters

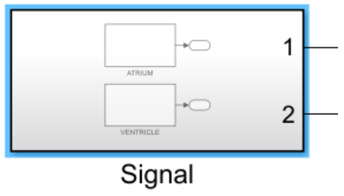
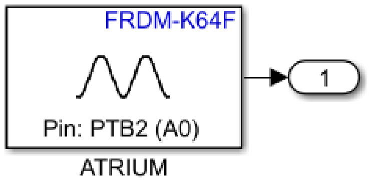
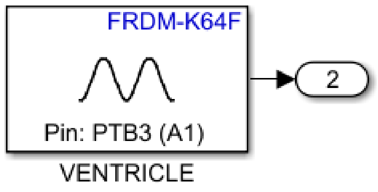
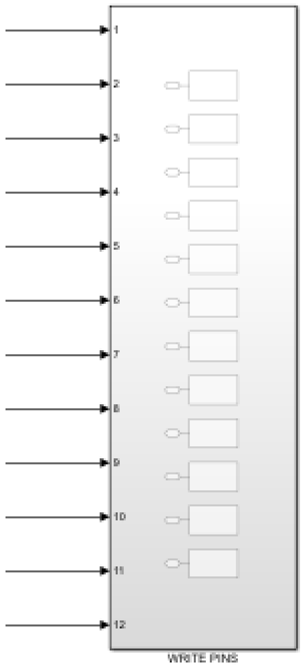
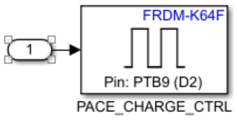
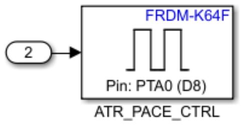
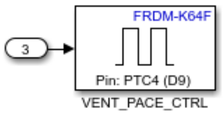
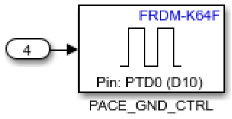
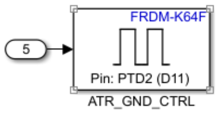
Variable	Type	Nominal Values
p_pacingMode	uint8	1 - AOO Mode 2 - VOO Mode 3 - AAI Mode 4 - VVI Mode 5 - DOO Mode 1 - AOOR Mode 3 - AAIR Mode 2 - VOOR Mode 4 - VVIR Mode 5 - DOOR Mode
p_lowrateLimit	uint8	60 ppm
p_vPaceAmp	single	5V
p_vPaceWidth	uint8	1ms
p_aPaceAmp	single	5V
p_aPaceWidth	uint8	1ms
p_upperateLimit	uint8	120 ppm
p_ARP	uint16	250ms
p_VRP	uint16	320ms
p_aSensitivity	single	0.75mV

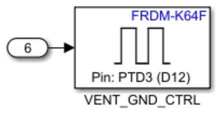
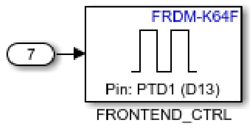
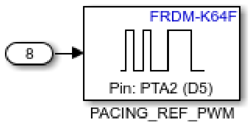
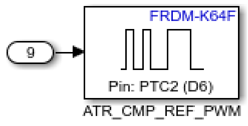
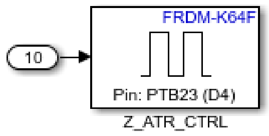
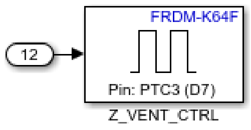
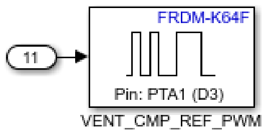
p_vSensitivity	single	2.5mV
p_fixedAVDelay	uint16	150ms
p_PVARP	uint16	250ms
p_hysterisis	uint8	0 (OFF)
p_rateSmoothing	uint8	0 (OFF)
p_maxSensorRate	uint8	120ms
p_activityThreshold	uint8	0 - Very Low 5 - Low 10 - Medium Low 15 - Medium (Nominal) 20 - Medium High 30 - High 35 - Very High
p_reactionTime	uint8	30s
p_responseFactor	unit8	8
p_recoveryTime	unit8	5min
p_UUID	uint8	Any non-zero number

Table 1: Programmable Parameters

Pins

Subsystem	**Pins**	
		Used in the sensing circuitry of the atrium. Outputs ON (HIGH) when signal voltage is higher than threshold voltage and OFF (LOW) otherwise (includes 5mV hysteresis).
		Used in the sensing circuitry of the ventricle. Outputs ON (HIGH) when signal voltage is higher than threshold voltage and OFF (LOW) otherwise (includes 5mV hysteresis).

	 This pin outputs the analog waveform of the atrium prior to fullwave rectification. This signal best represents what is actually happening in the heart in real-time. Use this analog output as data for any electrocardiogram outputs.  This pin outputs the analog waveform of the ventricle prior to fullwave rectification. This signal best represents what is actually happening in the heart in real-time. Use this analog output as data for any electrocardiogram outputs.
 1 2 3 4 5 6 7 8 9 10 11 12 WRITE PINS	 Used to start and stop the charging of the primary capacitor (C22). If ON (HIGH) → PWM charges C22 If OFF (LOW) → PWM disconnected from circuit  Used to discharge the primary capacitor through the atrium. Current flows through the switch if set to HIGH. If LOW there is no current flow. Pay attention to the direction at which current flows through the electrode.  Same functionality as in ATR PACE CTRL but for the ventricle.  To allow current to flow from the ring to the tip in either the atrium or the ventricle this pin must be HIGH since it controls the switch directly following the tip.  Used to connect the ATR RING OUT to GND. This functionality is used when discharging the blocking capacitor through the atrium to allow no charge buildup.

		Same functionality as in ATR RING OUT but for the ventricle.
		Used to activate the sensing circuitry. If ON (HIGH) →Sensing circuitry will output heart signal. If OFF (LOW) →Sensing circuitry is disconnected from patient (Green Connectors) and will output nothing.
		Used to charge the primary capacitor (C22) of the pacing circuit. The PWM voltage output by this pin saturates to 0-5V and will charge C22 is PACE CHARGE CTRL is HIGH
		Same functionality as in VENT CMP REF PWM but for the atrial action potential.
		This control allows the impedance circuit to be connected to the ring electrode of the atrium. It is used to analyze the impedance of the atrial electrode and the electrical connection between the atrial electrodes and the atrium itself. The output of this circuit is found at the Z SIGNAL pin.
		This control allows the impedance circuit to be connected to the ring electrode of the ventricle. Its use is identical to Z ATR CTRL but for the ventricle
		In order to establish a threshold for when the ventricular action potential should be sensed, this pin uses PWM to charge a capacitor that will sustain a constant voltage for comparison

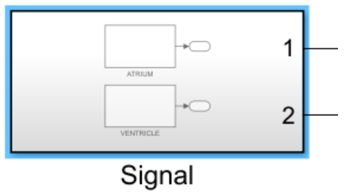
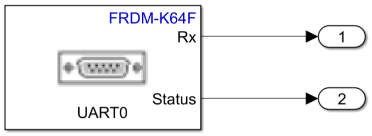
	 Used to receive serial communications from the DCM. The rxData signal received from the DCM will be 39 bytes long, and the status signal will change from its constant value of 32 to a value of 0 if a message of 39 bytes or greater is received.

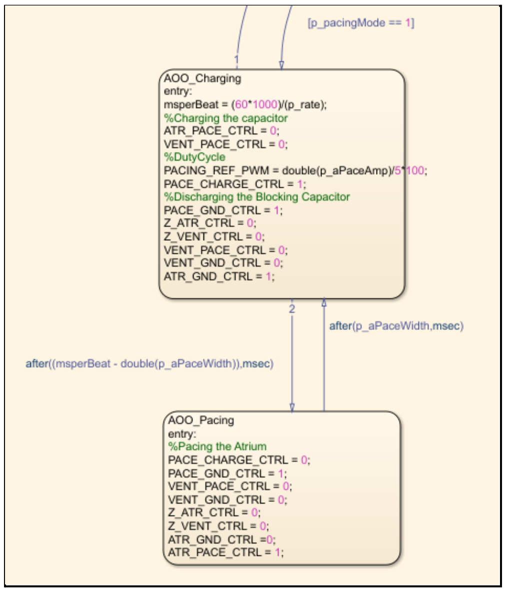
Table 2: Pins and Functionality

Design Decisions

Simulink was used to implement stateflows for a pacemaker in a permanent state, AOO, VOO, AAI, VVI, DOO, AOOR, VOOR, AAIR, VVIR and DOOR. The programmable parameters listed in the requirements document were used for the stateflow, the key components were the rate characteristics, such as the limits and delays, along with the pulse characteristics like width and regulated amplitude, with the chambers that were being placed as well. In accordance with the "nominal values'' column in the programmable parameters list, the values for ventricular amplitude and the atrial amplitude would be 5 V with the ventricular width and atrial width being 1 ms. The lower rate was 60 ppm and the upper limit rate was 120 ppm. In the sensing circuitry, it was decided that the logic would be to inhibit heart pacing of the atrium or ventricle if the detect pin outputs ON when the signal voltage is higher than threshold voltage. We chose this way of inhibiting the pacing so that it would be easier to identify when the Pacemaker was in the refractory period after a natural or artificial heart pulse. The pacemaker will be shaken by hand to simulate various speeds of activity; walking, jogging and running. The rate detection decision was based on the measured cardiac cycle lengths of the sensed rhythm. With the rate being evaluated on an interval by interval basis. We choose subsystems for hardware hiding as subsystems are known for this encapsulation. This restricts access to the information which is crucial for hardware hiding. The mode's pacing / charging were decided to be separate states as separation makes the sensing logic easier. This allows developers to mentally map states for current use and future additions to the map states as well. Finally, we chose to put all modes (AOO, VOO, AAI, VVI, DOO, AOOR, VOOR, AAIR, VVIR, DOOR) in one .slx file. It was chosen to be in one file to share the file between team members, and be easier to view all the states at once during the development stages. The development stage in the model driven development software cycle is a crucial stage. It has many benefits with great analysis and testing stages. Simulink is a design environment and is seen as a modern model driven tool suite.

Mode Descriptions

AOO/AOOR Mode

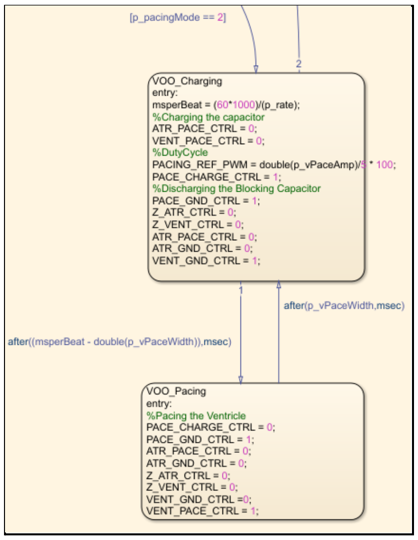


```
AOO_Charging
entry:
msperBeat = (60*1000)/(p_rate);
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_aPaceAmp)/5*100;
PACE_CHARGE_CTRL = 1;
%Discharging the Blocking Capacitor
PACE_GND_CTRL = 1;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
VENT_PACE_CTRL = 0;
VENT_GND_CTRL = 0;
ATR_GND_CTRL = 1;
```

```
AOO_Pacing
entry:
%Pacing the Atrium
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
VENT_PACE_CTRL = 0;
VENT_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_GND_CTRL =0;
ATR_PACE_CTRL = 1;
```

When the pacing mode is equal to 1, then the atrium will be paced without sensing the natural heart pulses. Note that when the board is shaken in the z-axis, the p_rate (pacing rate) will differ based on the activity level (AOOR mode), however if the board is set still, then p_rate (pacing rate) will be constant and equal to the LRL (AOO mode).

VOO/ VOOR Mode

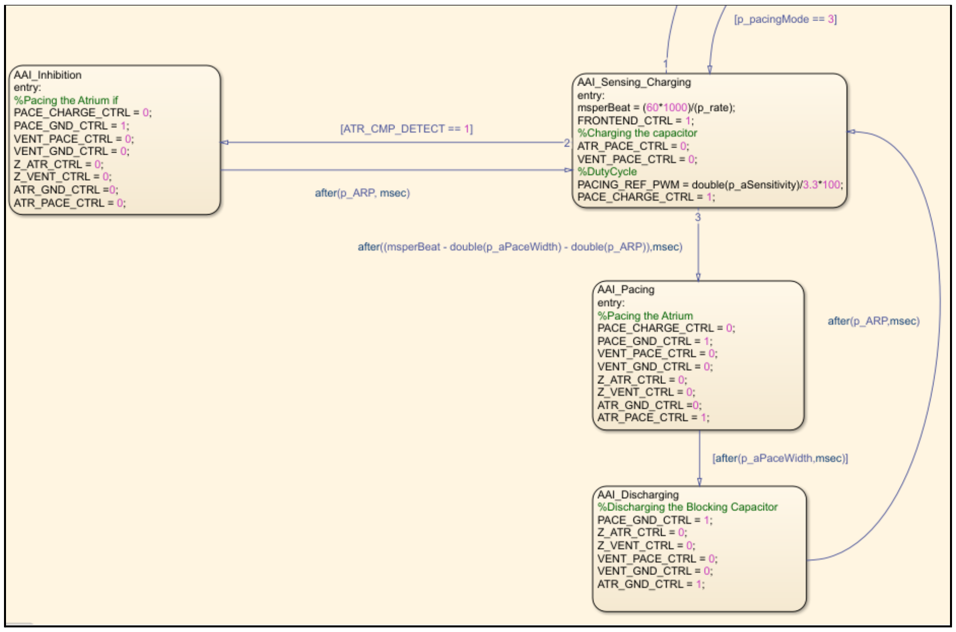


```
VOO_Charging
entry:
msperBeat = (60*1000)/(p_rate);
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_vPaceAmp)/5 * 100;
PACE_CHARGE_CTRL = 1;
%Discharging the Blocking Capacitor
PACE_GND_CTRL = 1;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
VENT_GND_CTRL = 1;
```

```
VOO_Pacing
entry:
%Pacing the Ventricle
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
VENT_GND_CTRL =0;
VENT_PACE_CTRL = 1;
```

When the pacing mode is equal to 2, then the ventricle will be paced without sensing the natural heart pulses. Note that when the board is shaken in the z-axis, the p_rate (pacing rate) will differ based on the activity level (VOOR mode), however if the board is set still, then p_rate (pacing rate) will be constant and equal to the LRL (VOO mode)

AAI/AAIR Mode



```
AAI_Inhibition
entry:
%Pacing the Atrium if
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
VENT_PACE_CTRL = 0;
VENT_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_GND_CTRL =0;
ATR_PACE_CTRL = 0;
```

```
AAI_Sensing_Charging
entry:
msperBeat = (60*1000)/(p_rate);
FRONTEND_CTRL = 1;
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_aSensitivity)/3.3*100;
PACE_CHARGE_CTRL = 1;
```

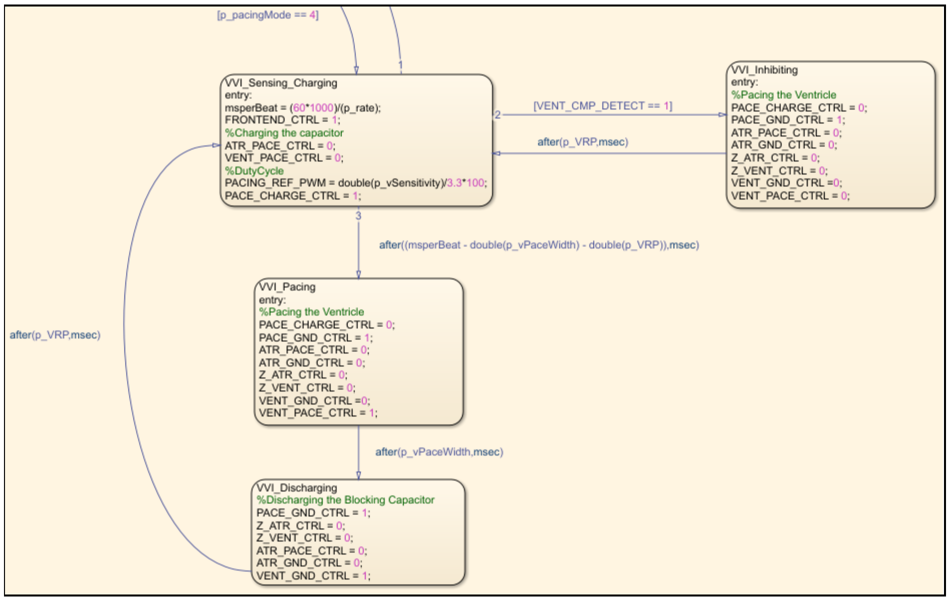
```
AAI_Pacing
entry:
%Pacing the Atrium
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
VENT_PACE_CTRL = 0;
VENT_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_GND_CTRL =0;
ATR_PACE_CTRL = 1;
```

```
AAI_Discharging
%Discharging the Blocking Capacitor
PACE_GND_CTRL = 1;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
VENT_PACE_CTRL = 0;
VENT_GND_CTRL = 0;
ATR_GND_CTRL = 1;
```

[p_pacingMode == 3]
1
2
3
[ATR_CMP_DETECT == 1]
after(p_ARP, msec)
after((msperBeat - double(p_aPaceWidth) - double(p_ARP)),msec)
after(p_ARP,msec)
[after(p_aPaceWidth,msec)]

When the pacing mode is equal to 3 the atrium will pace while sensing for natural heart pulses. If a natural heart pulse is detected the atrium won't pulse during the cycle. Note that when the board is shaken in the z-axis, the p_rate (pacing rate) will differ based on the activity level (AAIR mode), however if the board is set still, then p_rate (pacing rate) will be constant and equal to the LRL (AAI mode).

VVI/VVIR Mode



```
VVI_Sensing_Charging
entry:
msperBeat = (60*1000)/(p_rate);
FRONTEND_CTRL = 1;
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_vSensitivity)/3.3*100;
PACE_CHARGE_CTRL = 1;
```

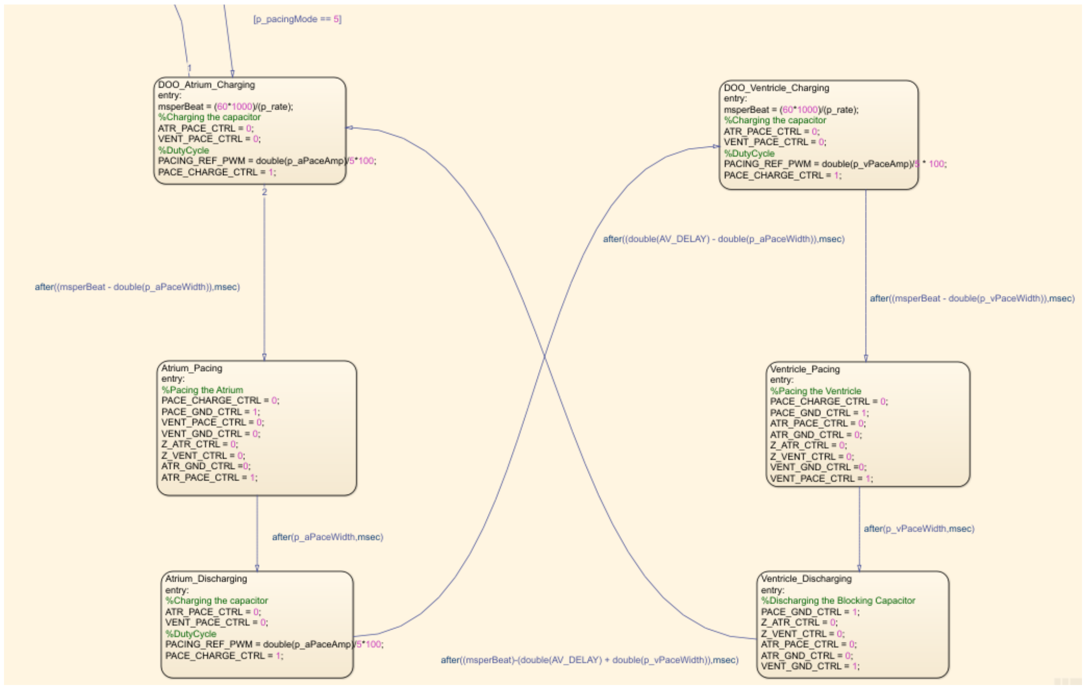
```
VVI_Inhibiting
entry:
%Pacing the Ventricle
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
VENT_GND_CTRL =0;
VENT_PACE_CTRL = 0;
```

```
VVI_Pacing
entry:
%Pacing the Ventricle
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
VENT_GND_CTRL =0;
VENT_PACE_CTRL = 1;
```

```
VVI_Discharging
%Discharging the Blocking Capacitor
PACE_GND_CTRL = 1;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
VENT_GND_CTRL = 1;
```

When the pacing mode is equal to 4 the ventricle will pace while sensing for natural heart pulses. If a natural heart pulse is detected the ventricle won't pulse during the cycle. Note that when the board is shaken in the z-axis, the p_rate (pacing rate) will differ based on the activity level (VVIR mode), however if the board is set still, then p_rate (pacing rate) will be constant and equal to the LRL (VVI mode).

DOO/DOOR Mode


[p_pacingMode == 5]
1
2
after((msperBeat - double(p_aPaceWidth)),msec)
after(p_aPaceWidth,msec)
after((double(AV_DELAY) - double(p_aPaceWidth)),msec)
after((msperBeat - double(p_vPaceWidth)),msec)
after(p_vPaceWidth,msec)
after((msperBeat)-(double(AV_DELAY) + double(p_vPaceWidth)),msec)

```
DOO_Atrium_Charging
entry:
msperBeat = (60*1000)/(p_rate);
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_aPaceAmp)/5*100;
PACE_CHARGE_CTRL = 1;
```

```
Atrium_Pacing
entry:
%Pacing the Atrium
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
VENT_PACE_CTRL = 0;
VENT_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_GND_CTRL =0;
ATR_PACE_CTRL = 1;
```

```
Atrium_Discharging
entry:
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_aPaceAmp)/5*100;
PACE_CHARGE_CTRL = 1;
```

```
DOO_Ventricle_Charging
entry:
msperBeat = (60*1000)/(p_rate);
%Charging the capacitor
ATR_PACE_CTRL = 0;
VENT_PACE_CTRL = 0;
%DutyCycle
PACING_REF_PWM = double(p_vPaceAmp)/5 * 100;
PACE_CHARGE_CTRL = 1;
```

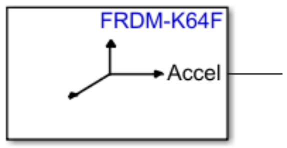
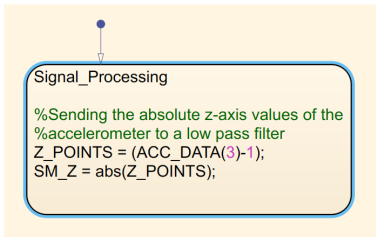
```
Ventricle_Pacing
entry:
%Pacing the Ventricle
PACE_CHARGE_CTRL = 0;
PACE_GND_CTRL = 1;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
VENT_GND_CTRL =0;
VENT_PACE_CTRL = 1;
```

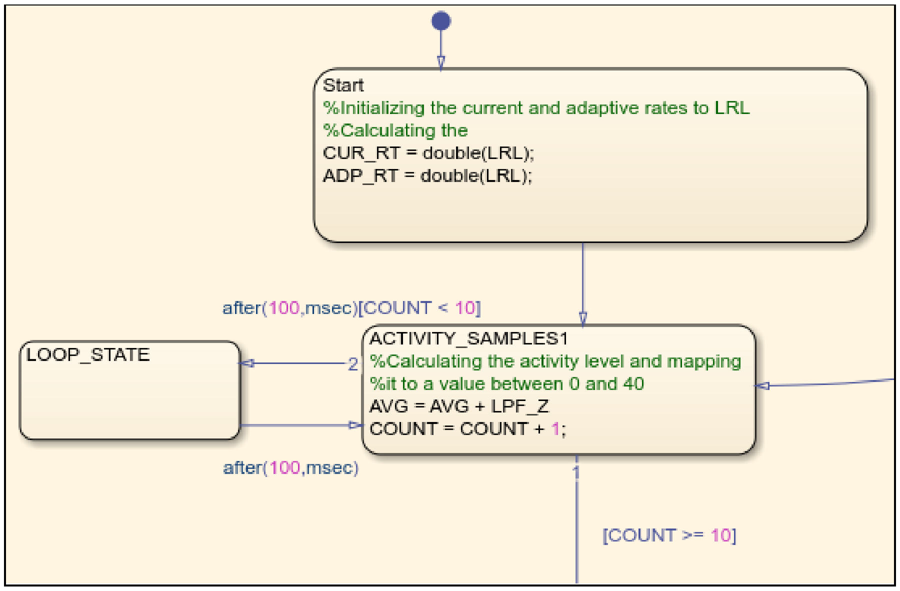
```
Ventricle_Discharging
entry:
%Discharging the Blocking Capacitor
PACE_GND_CTRL = 1;
Z_ATR_CTRL = 0;
Z_VENT_CTRL = 0;
ATR_PACE_CTRL = 0;
ATR_GND_CTRL = 0;
VENT_GND_CTRL = 1;
```

When the pacing mode is equal to 5 the atrium and ventricle will pace without sensing for a natural heart pulse. The atrium and ventricle pulses are phase shifted by a constant value, p_fixedDelay, which

is programmably controlled through the DCM. The logic begins with the atrium charge and discharge followed by the ventricle charge and discharge with an offset of p_fixedDelay. Note that when the board is shaken in the z-axis, the p_rate (pacing rate) will differ based on the activity level (DOOR mode), however if the board is set still, then p_rate (pacing rate) will be constant and equal to the LRL (DOO mode). This applies for both the atrium and the ventricle.

Rate Adaptive Logic

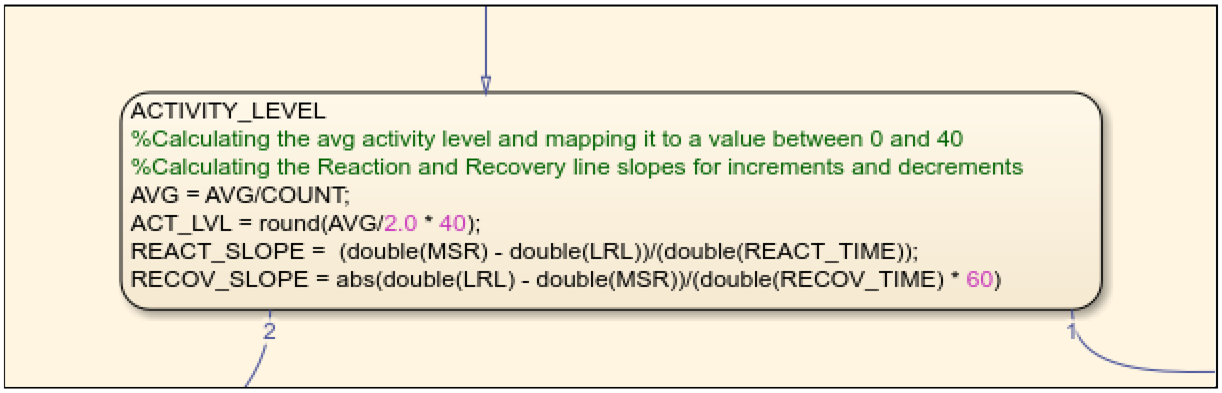
FXOS8700 6-Axes Sensor		The 6-Axes Sensor measures the acceleration in the x,y and z direction.
Signal Processing Logic	 Signal_Processing %Sending the absolute z-axis values of the %accelerometer to a low pass filter Z_POINTS = (ACC_DATA(3)-1); SM_Z = abs(Z_POINTS);	The accelerometer data for only the z-axis is collected, since activity detection is tested by moving the board up and down. Moreover, absolute value is taken since the direction of acceleration does does not matter
Low Pass Filter		A low pass filter was used to filter out high frequency components of the acceleration,
Rate Adaptive		



```
Start
%Initializing the current and adaptive rates to LRL
%Calculating the
CUR_RT = double(LRL);
ADP_RT = double(LRL);
```

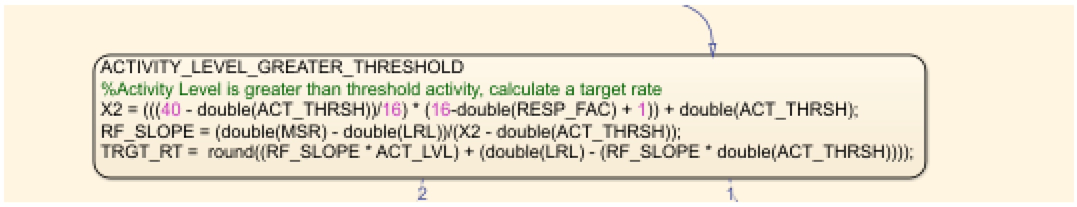
```
ACTIVITY_SAMPLES1
%Calculating the activity level and mapping
%it to a value between 0 and 40
AVG = AVG + LPF_Z
COUNT = COUNT + 1;
```

When flashing the board for the first time, the current pacing rate and the adaptive pacing rate is set to the Lower Rate Limit. To accurately calculate the activity level, 10 samples of the filtered acceleration are taken



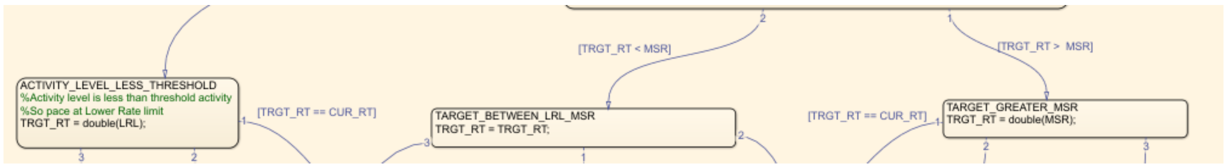
```
ACTIVITY_LEVEL
%Calculating the avg activity level and mapping it to a value between 0 and 40
%Calculating the Reaction and Recovery line slopes for increments and decrements
AVG = AVG/COUNT;
ACT_LVL = round(AVG/2.0 * 40);
REACT_SLOPE =  (double(MSR) - double(LRL))/(double(REACT_TIME));
RECOV_SLOPE = abs(double(LRL) - double(MSR))/(double(RECOV_TIME) * 60)
```

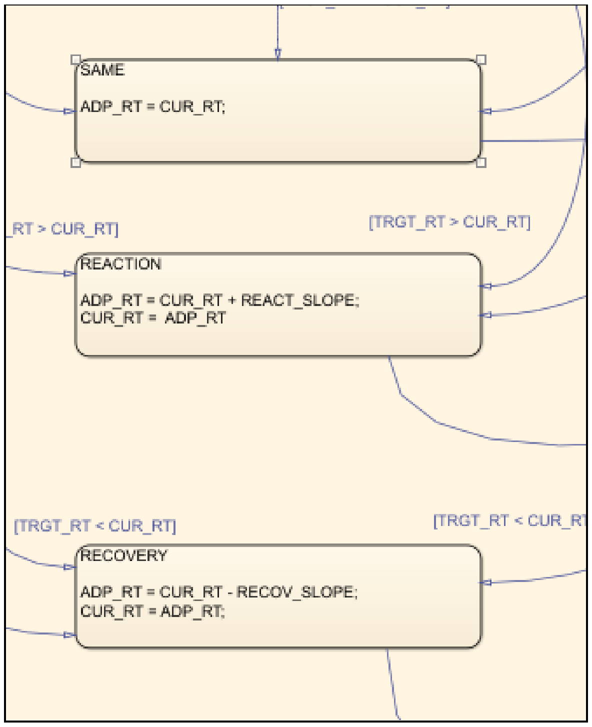
The samples are averaged to get the acceleration value (0- 2g) and this value is scaled to an activity level between 0 nad 40. In addition, the reaction slope and the recovery slope (BPM/s) for the incremental/decremental step used later in adaptive rate calculation is determined.

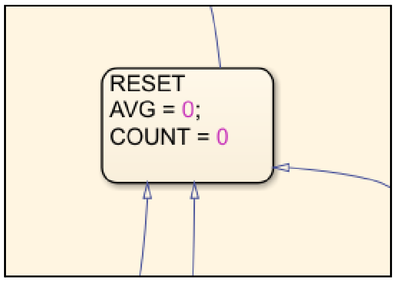


```
ACTIVITY_LEVEL_GREATER_THRESHOLD
%Activity Level is greater than threshold activity, calculate a target rate
X2 = (((40 - double(ACT_THRSH))/16) * (16-double(RESP_FAC) + 1)) + double(ACT_THRSH);
RF_SLOPE = (double(MSR) - double(LRL))/(X2 - double(ACT_THRSH));
TRGT_RT =  round((RF_SLOPE * ACT_LVL) + (double(LRL) - (RF_SLOPE * double(ACT_THRSH))));
```

If the activity level is greater than the activity threshold parameter, then a target rate is calculated based on the response factor parameter. If the response factor is low, then the slope calculated is low, however if the response factor is high, then the slope calculated is also high.

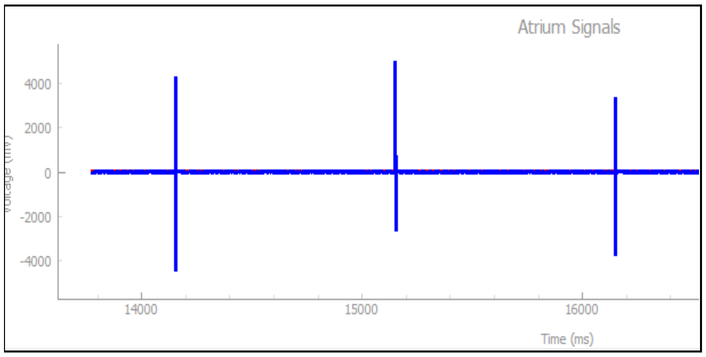
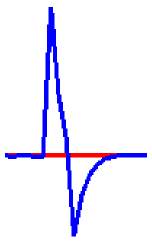
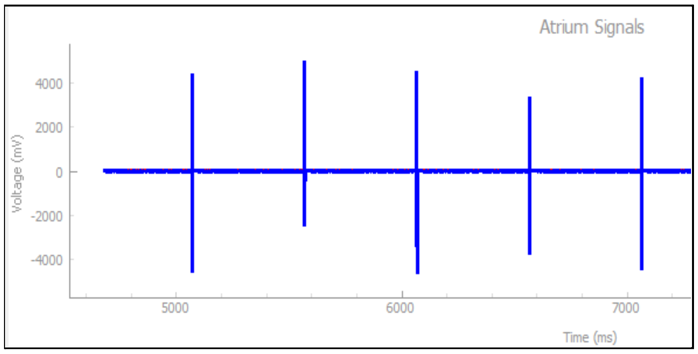
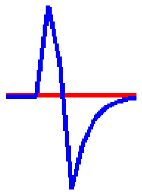
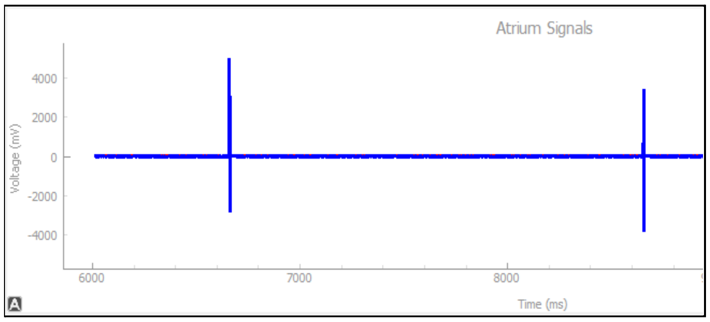
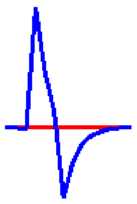
 ACTIVITY_LEVEL_LESS_THRESHOLD %Activity level is less than threshold activity %So pace at Lower Rate limit TRGT_RT = double(LRL); [TRGT_RT == CUR_RT] [TRGT_RT < MSR] TARGET_BETWEEN_LRL_MSR TRGT_RT = TRGT_RT; [TRGT_RT > MSR] [TRGT_RT == CUR_RT] TARGET_GREATER_MSR TRGT_RT = double(MSR);	
If the activity level is less than the threshold, then the target pacing rate is lower rate limit, if the target rate is in between LRL and MSR, then the target pacing rate is equal to the calculated target rate. However, if the calculated target exceeds the MSR, then the target rate is reset to MSR.	

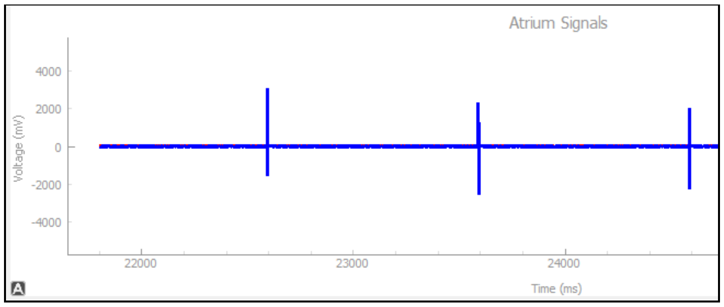
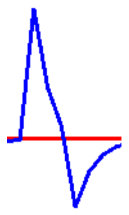
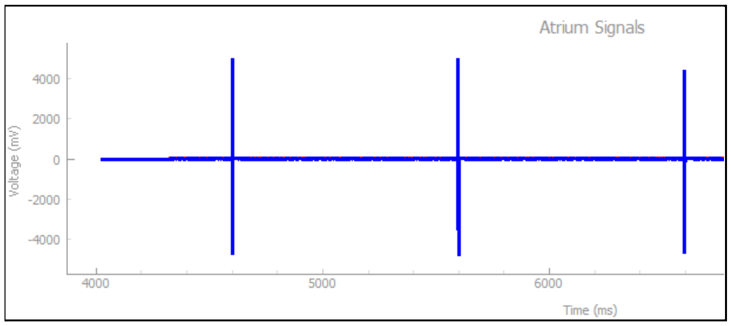
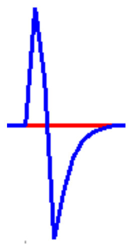
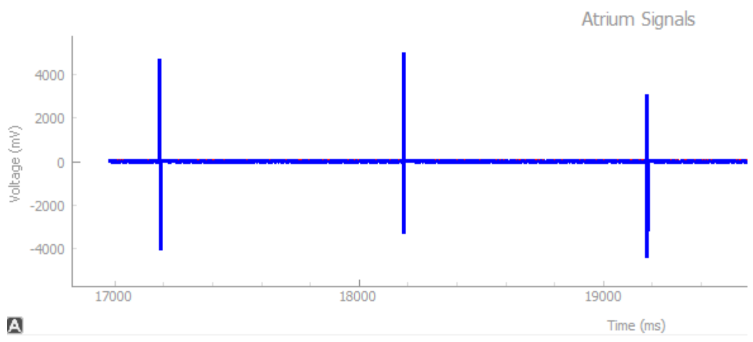
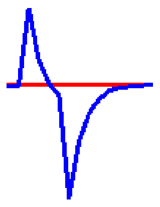
 SAME
ADP_RT = CUR_RT;
[TRGT_RT > CUR_RT]
REACTION
ADP_RT = CUR_RT + REACT_SLOPE;
CUR_RT = ADP_RT
[TRGT_RT < CUR_RT]
RECOVERY
ADP_RT = CUR_RT - RECOV_SLOPE;
CUR_RT = ADP_RT;
	If the current rate is equal to the target rate calculated, then adaptive pacing rate sent to the pacemaker logic is equal to the current rate. If the target rate is greater than the current rate, then the pacing rate is equal to the current rate plus the incremental rate (react slope calculated before). Similarly, if the target rate is less than the current rate, then the pacing rate is equal to the current rate minus the decremental rate (recovery slope calculated before). The current rate is updated to the pacing rate, so that when the activity level is calculated in the next cycle the decision is made based on the new current rate.

	Once the pacing rate is outputted, and the current rate is updated, the avg and count local variables are reset for calculating the next average acceleration, activity level and target rate.

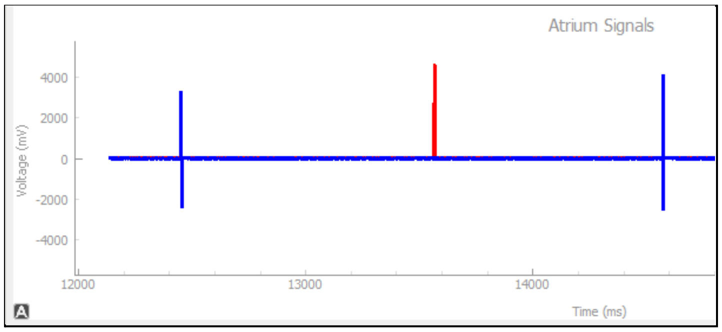
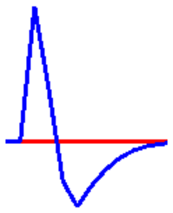
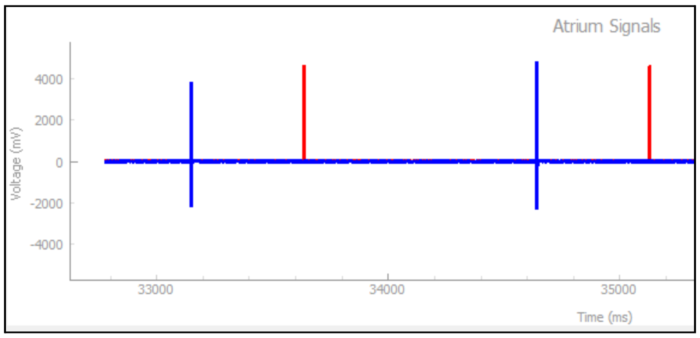
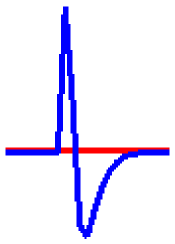
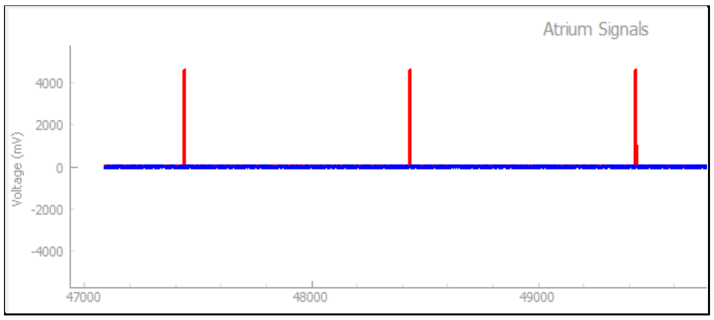
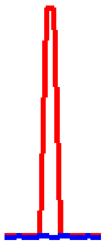
Testing (Non Rate Adaptive Modes)

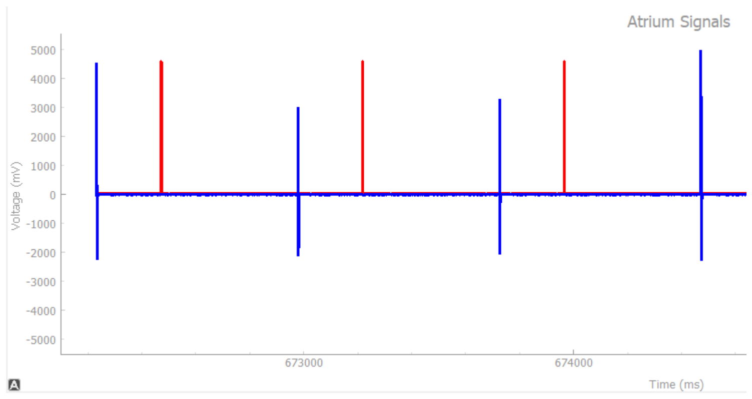
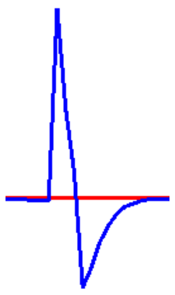
AOO Mode		
Parameters	**HeartView Atrium Pacing**	**Pulse**

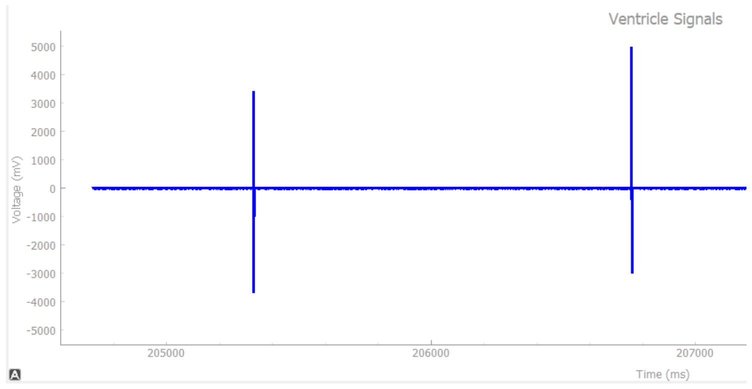
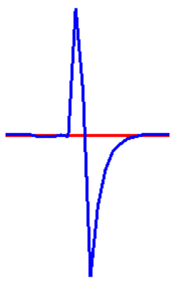
Normal Heart Rate Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms		
The above test ensures that the AOO mode works as intended with the programmable parameters set to its nominal values. From the Heartview simulation, we can see that the pulses are approximately 3500mV (3.5V), with the pulse occurring every 1000ms approximately (60 BPM). **Result: PASS**		
Fast Heart Rate Rate: 120BPM Pulse Amplitude: 3.5V Pulse Width: 1ms		
The above test ensures that the AOO mode works as intended with the programmable parameters set to its nominal values, except for the heart rate, which is set to 120 BPM. From the Heartview simulation, we see that the pulses are occurring every 500ms approximately as intended (120 BPM). **Result: PASS**		
Slow Heart Rate Rate: 30BPM Pulse Amplitude: 3.5V Pulse Width: 1ms		
The above test ensures that the AOO mode works as intended with the programmable parameters set to its nominal values, except for the heart rate, which is set to 30 BPM. From the Heartview simulation, we can see that the pulses are occurring every 2000ms approximately (30 BPM) as intended. **Result: PASS**		

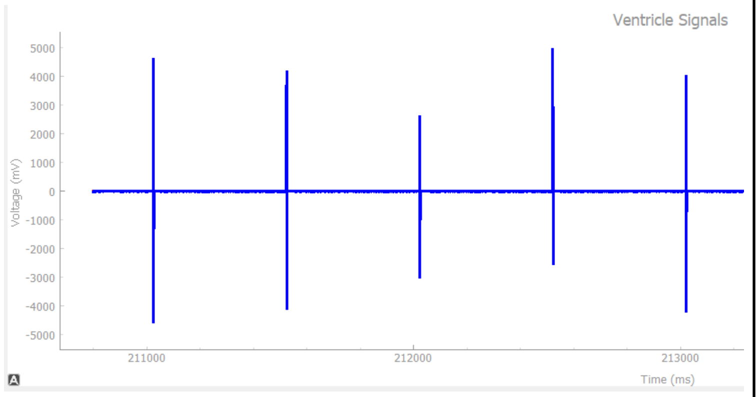
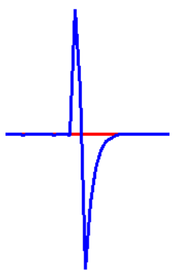
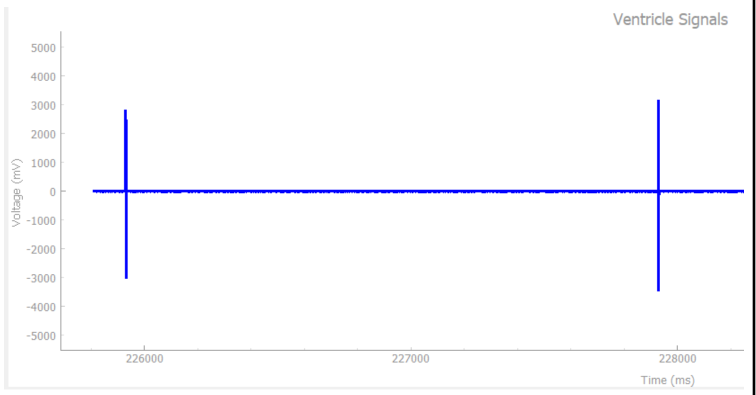
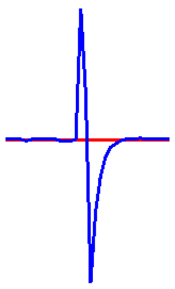
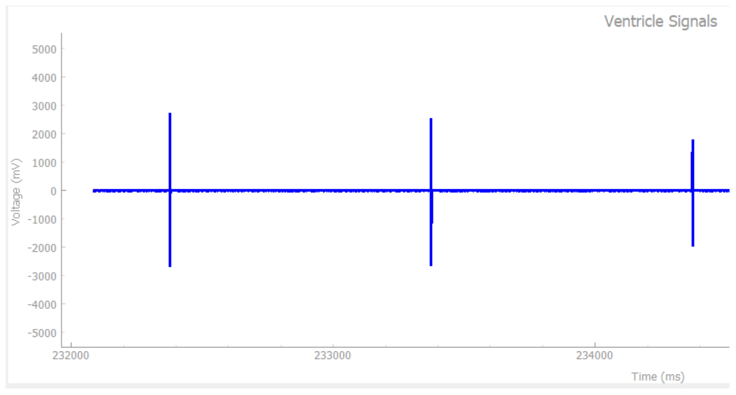
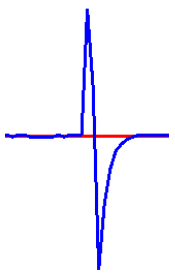
Decrease Voltage Rate: 60BPM Pulse Amplitude: 2V Pulse Width: 1ms		
The above test ensures that the AOO mode works as intended with the programmable parameters set to its nominal values, except for voltage which is decreased to 2V. From the Heartview simulation, we can see that the pulses are approximately 2000mV (2V), with the pulse occurring every 1000ms approximately (60 BPM). **Result: PASS**		
Increase Voltage Rate: 60BPM Pulse Amplitude: 4.5V Pulse Width: 1ms		
The above test ensures that the AOO mode works as intended with the programmable parameters set to its nominal values, except for voltage which is increased to 4V. From the Heartview simulation, we can see that the pulses are approximately 4000mV (4V), with the pulse occurring every 1000ms approximately (60 BPM). **Result: PASS**		
Increase Pulse Width Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 2ms		
The above test ensures that the AOO mode works as intended with the programmable parameters set to its nominal values, except for the pulse width which is increased to 2ms. From the Heartview simulation, we can see that the pulses are approximately 3500mV (2V), with the pulse occurring every 1000ms approximately (60 BPM). When we look at a particular pulse we can see that it is wider compared to the tests performed above. **Result: PASS**		

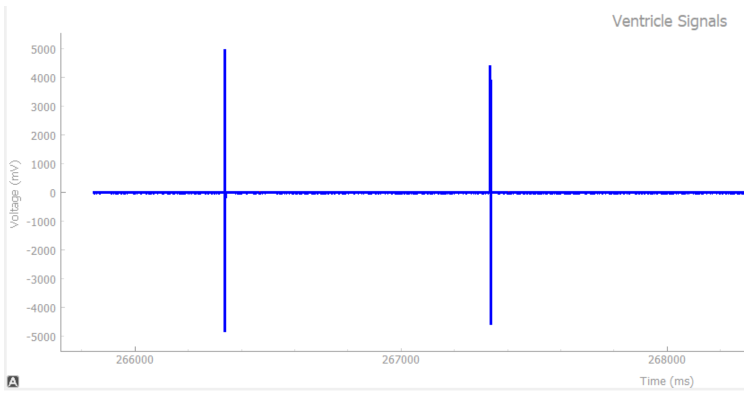
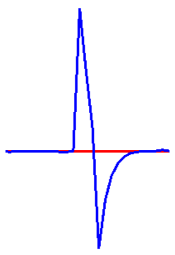
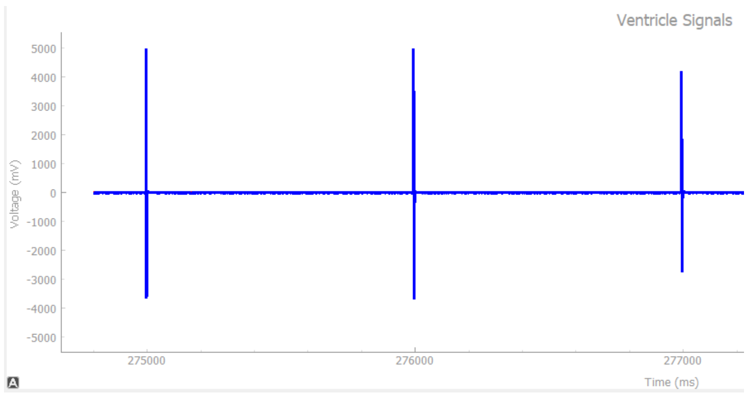
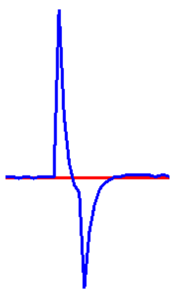
AAI Mode

Parameters	HeartView Atrium Pacing	Pulse
Normal Heart Rate **Pacemaker** Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms ARP: 250ms **Heart** Heart Rate: 30BPM Pulse Width: 1ms		
The above test ensures that the AAI mode works as intended with the pacemaker programmable parameters set to its nominal values, and the Heart rate set to a slow pulse of 30 BPM. From the Heartview simulation we can see that the pacemaker paces the heart (blue pulse) every 1000 ms, but inhibits a pace when it detects a natural pulse from the heart (red pulse). Therefore we should get an alternating pattern of pacemaker pulse followed by heart pulse. **Result: PASS**		
Faster Natural Heart **Pacemaker** Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms ARP: 250ms **Heart** Heart Rate: 40BPM Pulse Width: 1ms		
The above test ensures that the AAI mode works as intended with the pacemaker programmable parameters set to its nominal values, and the Heart rate set to a slightly faster pulse of 40 BPM. Expected behaviour is alternating pacemaker and heart pulse, but the heart pulse should appear closer to the pacemaker pulse because the heart rate is faster. From the simulation results we can see that we indeed see the expected behaviour. **Result: PASS**		
Natural Heart Rate Equal to Pacemaker **Pacemaker** Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms ARP: 370ms **Heart** Heart Rate: 60BPM Pulse Width: 1ms		

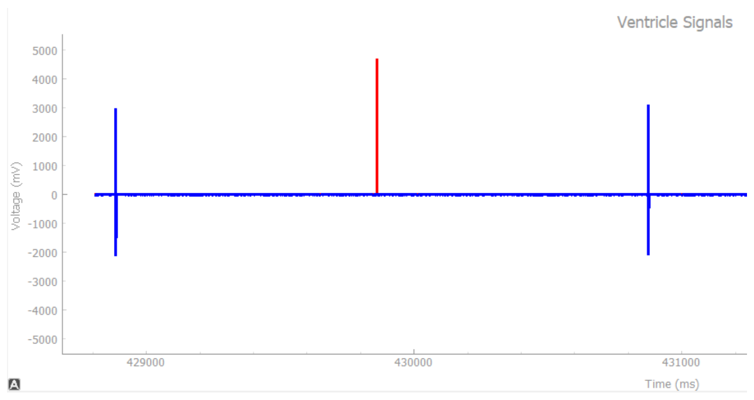
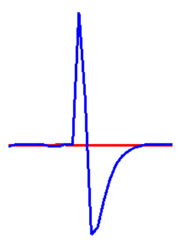
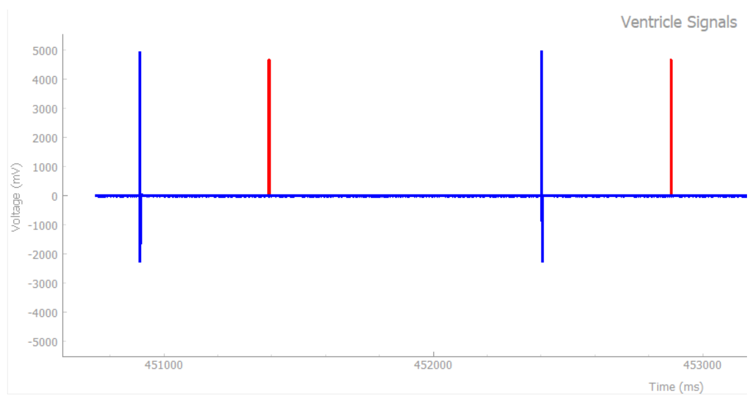
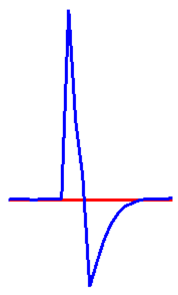
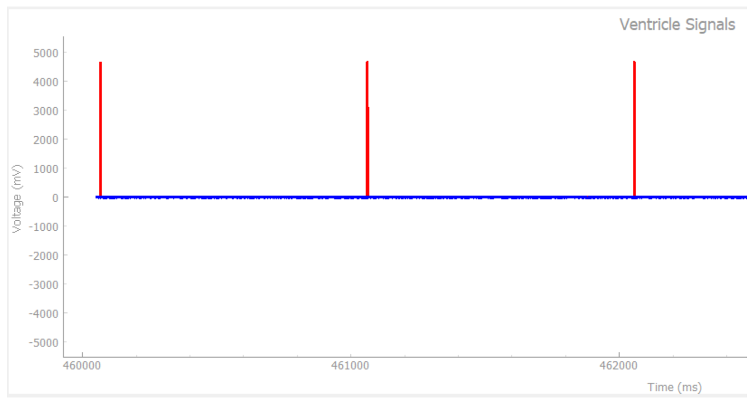
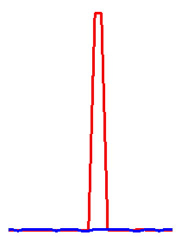
The above test ensures that when the heart is beating normally (at 60 BPM), the pacemaker does not pace the heart. **Result: PASS**		
Natural Heart and Pacemaker Increased Heart Rate **Pacemaker** Rate: 120BPM Pulse Amplitude: 3.5V Pulse Width: 1ms ARP: 150ms **Heart** Heart Rate: 80BPM Pulse Width: 1ms		
The above test ensures that when the heart is beating faster than normal (at 80BPM), the pacemaker paces since the Lower Rate Limit is above the normal heart rate. **Result: PASS**		

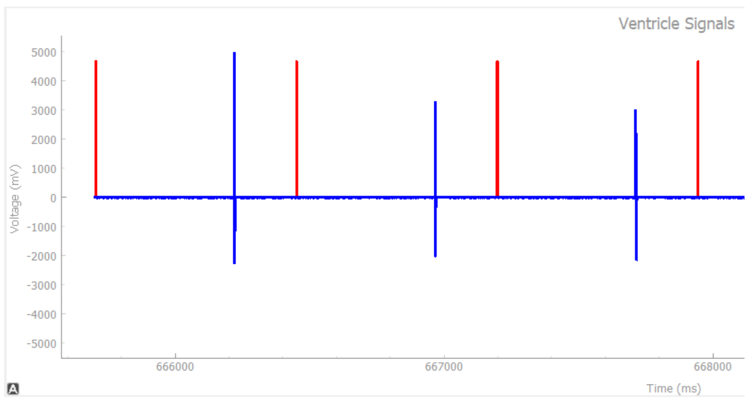
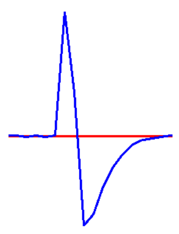
VOO Mode		
Parameters	**HeartView Atrium Pacing**	**Pulse**
Normal Heart Rate Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms		
The above test ensures that the VOO mode works as intended with the programmable parameters set to its nominal values. From the Heartview simulation, we can see that the pulses are approximately 3500mV (3.5V), with the pulse occurring every 1000ms approximately (60 BPM). **Result: PASS**		

Fast Heart Rate Rate: 120BPM Pulse Amplitude: 3.5V Pulse Width: 1ms		
The above test ensures that the VOO mode works as intended with the programmable parameters set to its nominal values, except for the heart rate, which is set to 120 BPM. From the Heartview simulation, we can see that the pulses are occurring every 500ms approximately as intended (120 BPM). **Result: PASS**		
Slow Heart Rate Rate: 30BPM Pulse Amplitude: 3.5V Pulse Width: 1ms		
The above test ensures that the VOO mode works as intended with the programmable parameters set to its nominal values, except for the heart rate, which is set to 30 BPM. From the Heartview simulation, we can see that the pulses are occurring every 2000ms approximately (30 BPM) as intended. **Result: PASS**		
Decrease Voltage Rate: 60BPM Pulse Amplitude: 2V Pulse Width: 1ms		
The above test ensures that the VOO mode works as intended with the programmable parameters set to its nominal values, except for voltage which is decreased to 2V. From the Heartview simulation, we can see that the pulses are approximately 2000mV (2V), with the pulse occurring every 1000ms		

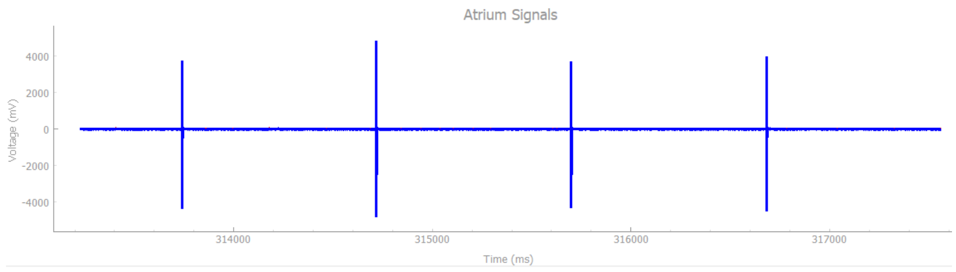
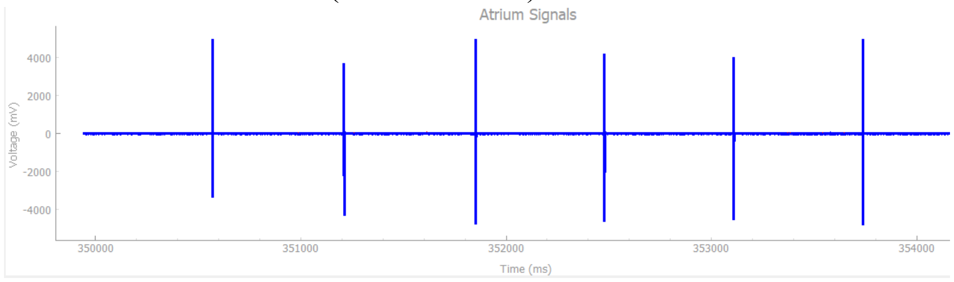
approximately (60 BPM). **Result: PASS**		
Increase Voltage Rate: 60BPM Pulse Amplitude: 4.5V Pulse Width: 1ms		
The above test ensures that the VOO mode works as intended with the programmable parameters set to its nominal values, except for voltage which is increased to 4V. From the Heartview simulation, we can see that the pulses are approximately 4000mV (4V), **Result: PASS**		
Increase Pulse Width Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 2ms		
The above test ensures that the VOO mode works as intended with the programmable parameters set to its nominal values, except for the pulse width which is increased to 2ms. From the Heartview simulation, we can see that the pulses are approximately 3500mV (2V), with the pulse occurring every 1000ms approximately (60 BPM). When we look at a particular pulse we can see that it is wider compared to the tests performed above. **Result: PASS**		

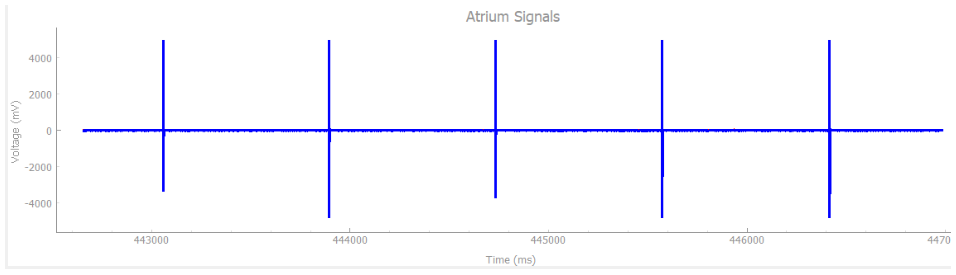
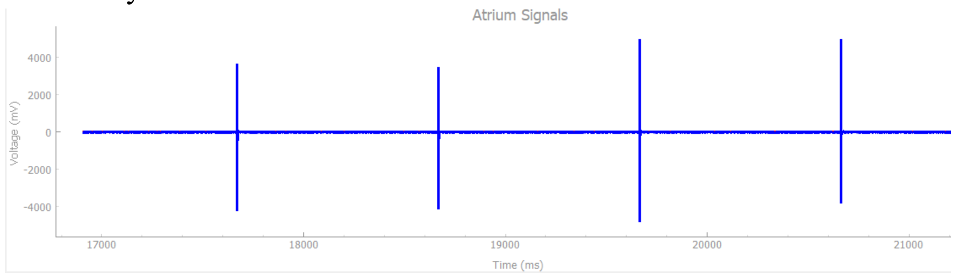
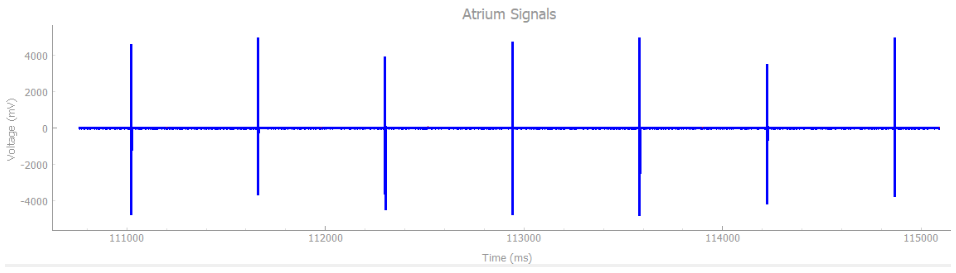
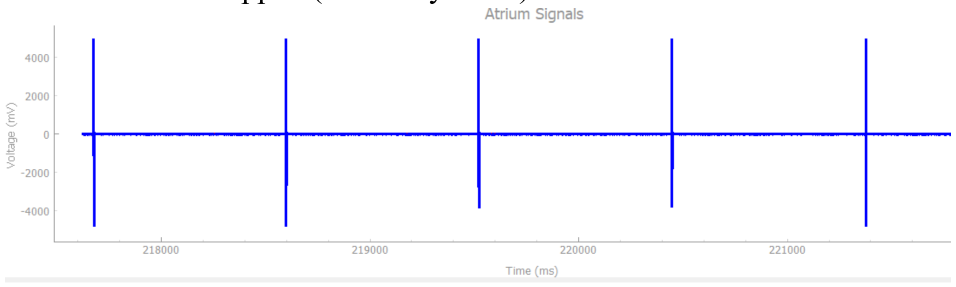
VVI Mode		
Parameters	**HeartView Atrium Pacing**	**Pulse**

Normal Heart Rate **Pacemaker** Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms VRP: 320ms **Heart** Heart Rate: 30BPM Pulse Width: 1ms		
The above test ensures that the VVI mode works as intended with the pacemaker programmable parameters set to its nominal values, and the Heart rate set to a slow pulse of 30 BPM. From the Heartview simulation we can see that the pacemaker paces the heart (blue pulse) every 1000 ms, but inhibits a pace when it detects a natural pulse from the heart (red pulse). Therefore we should get an alternating pattern of pacemaker pulse followed by heart pulse. **Result: PASS**		
Faster Natural Heart **Pacemaker** Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms VRP: 320ms **Heart** Heart Rate: 40BPM Pulse Width: 1ms		
The above test ensures that the VVI mode works as intended with the pacemaker programmable parameters set to its nominal values, and the Heart rate set to a slightly faster pulse of 40 BPM. Expected behaviour is alternating pacemaker and heart pulse, but the heart pulse should appear closer to the pacemaker pulse because the heart rate is faster. From the simulation results we can see that we indeed see the expected behaviour. **Result: PASS**		
Natural Heart Rate Equal to Pacemaker **Pacemaker** Rate: 60BPM Pulse Amplitude: 3.5V Pulse Width: 1ms VRP: 320ms **Heart** Heart Rate: 60BPM Pulse Width: 1ms		

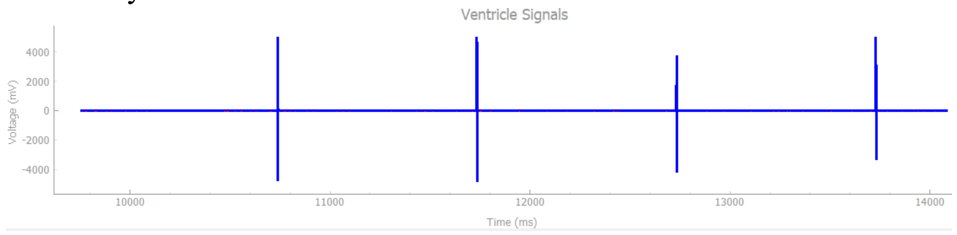
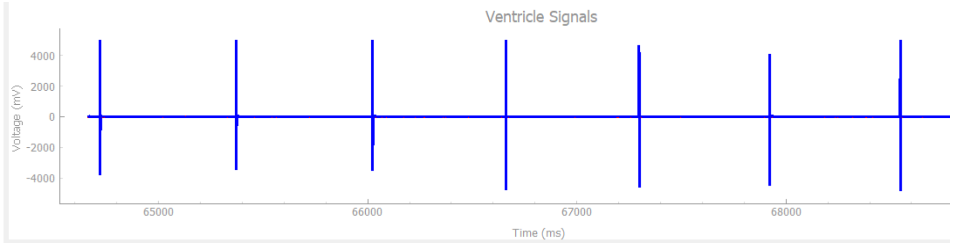
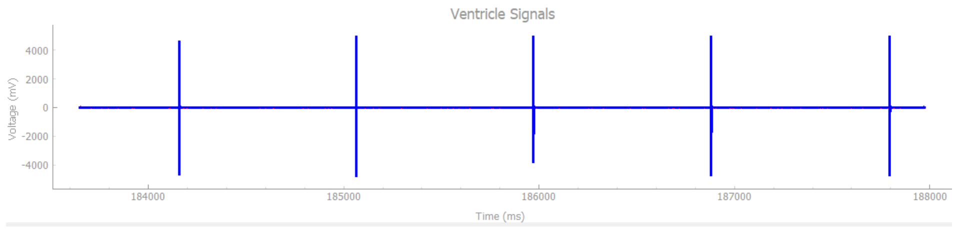
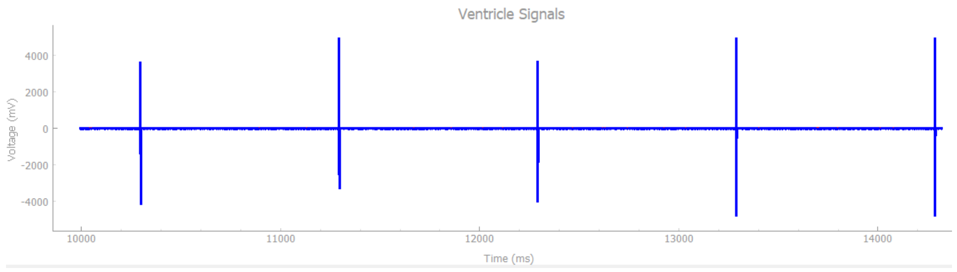
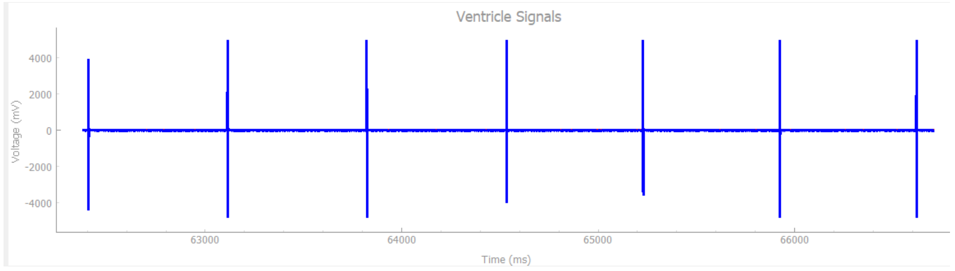
The above test ensures that when the heart is beating normally (at 60 BPM), the pacemaker does not pace the heart. **Result: PASS**		
Natural Heart and Pacemaker Increased Heart Rate **Pacemaker** Rate: 120BPM Pulse Amplitude: 3.5V Pulse Width: 1ms VRP: 150ms **Heart** Heart Rate: 80BPM Pulse Width: 1ms		
The above test ensures that when the heart is beating faster than normal (at 80BPM), the pacemaker paces since the Lower Rate Limit is above the normal heart rate. **Result: PASS**		

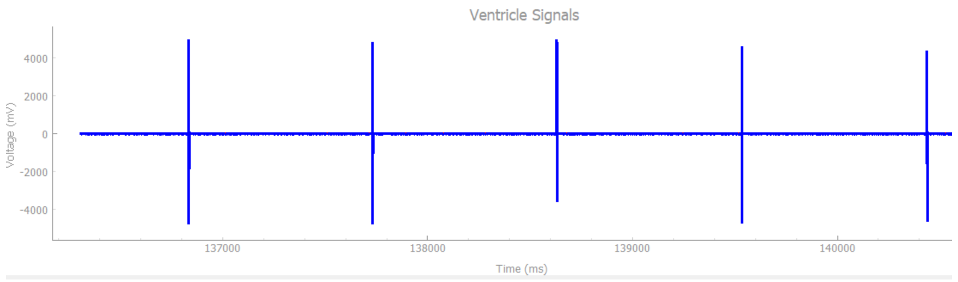
Testing (Rate Adaptive Modes)

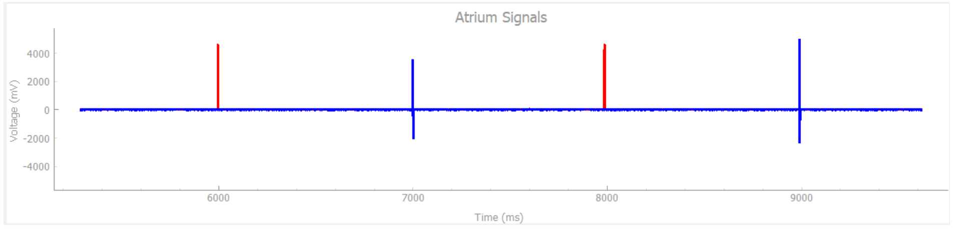
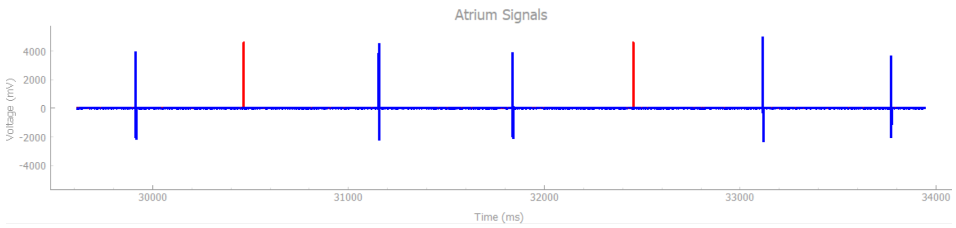
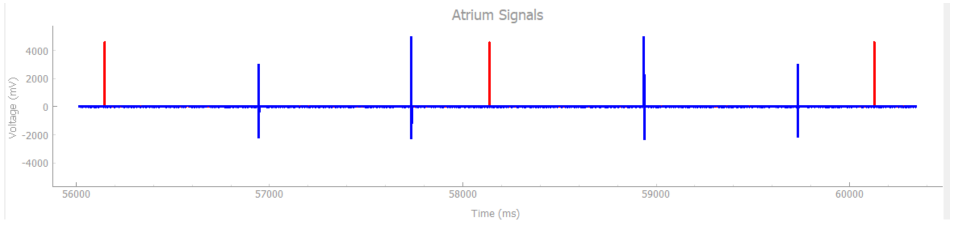
AOOR Mode	
Parameters	**HeartView Atrium Pacing**
Low Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 5 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)

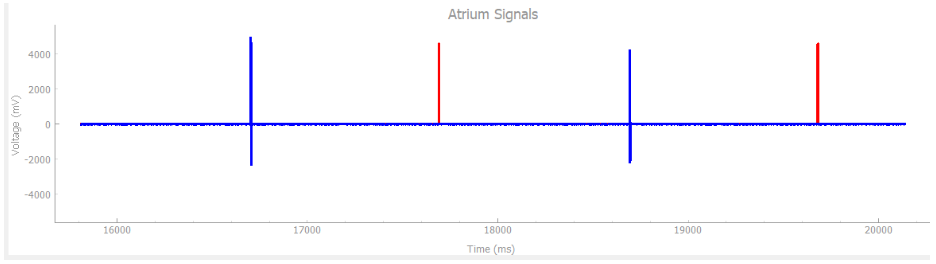
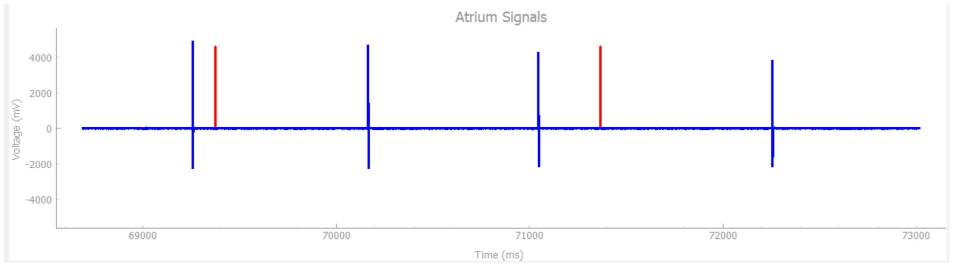
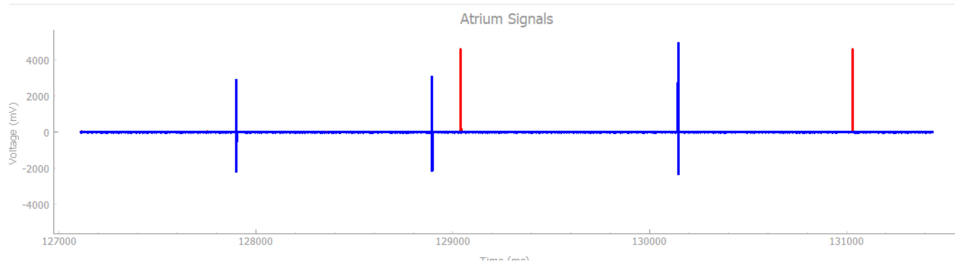
	 Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards. **Result: PASS**
High Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 30 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)  Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards. **Result: PASS**

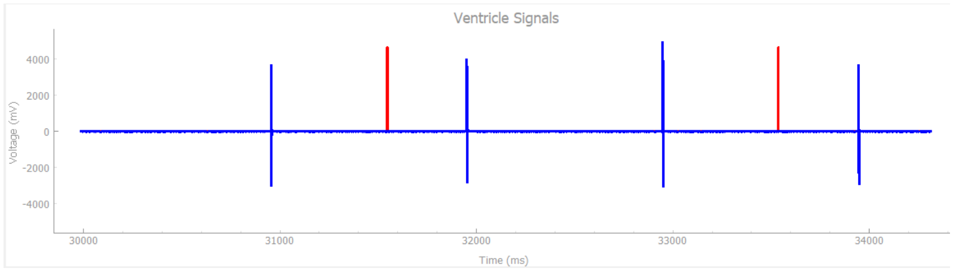
VOOR Mode	
Parameters	**HeartView Atrium Pacing**

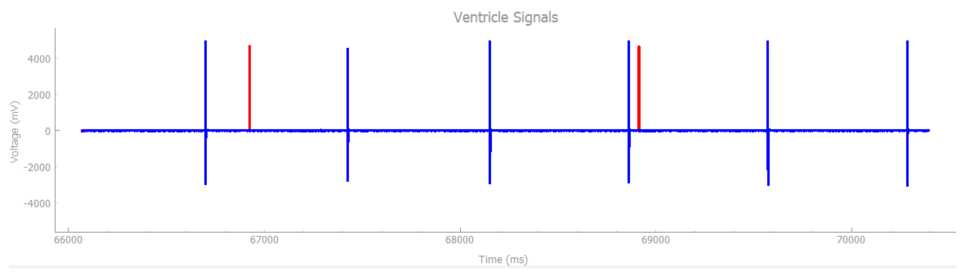
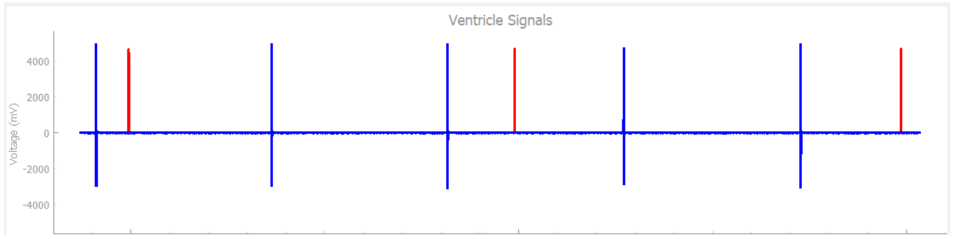
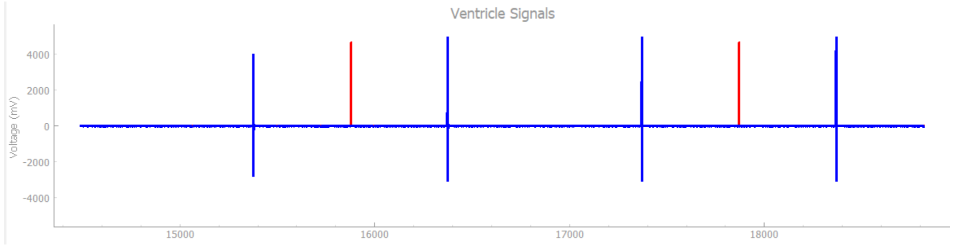
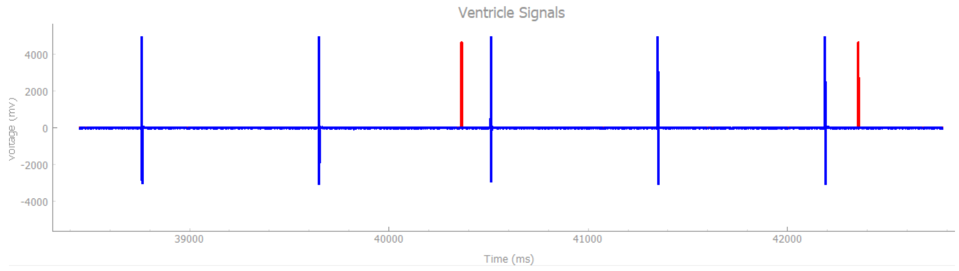
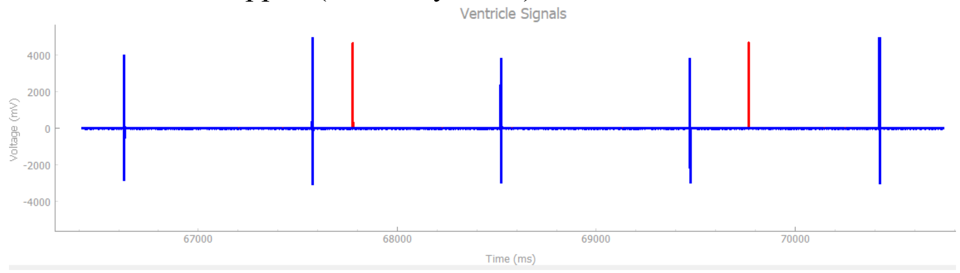
Low Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 5 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)  Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards. **Result: PASS**
High Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 30 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)

	 Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards. **Result: PASS**

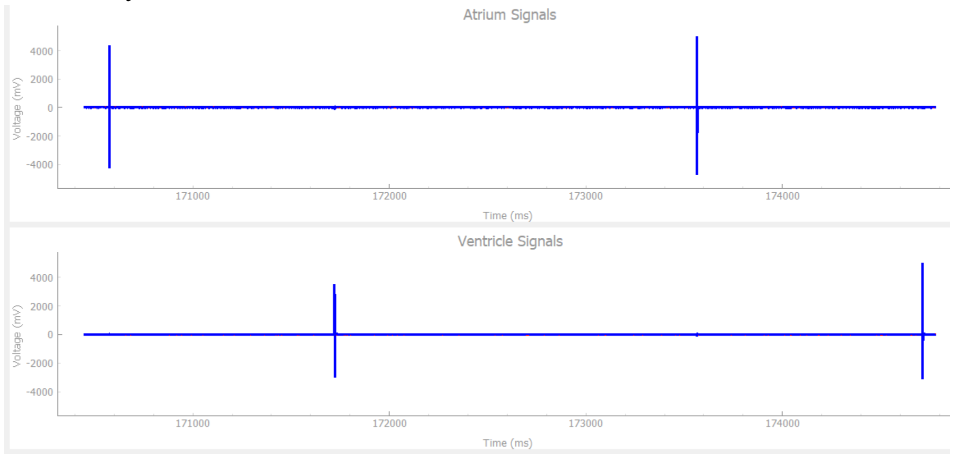
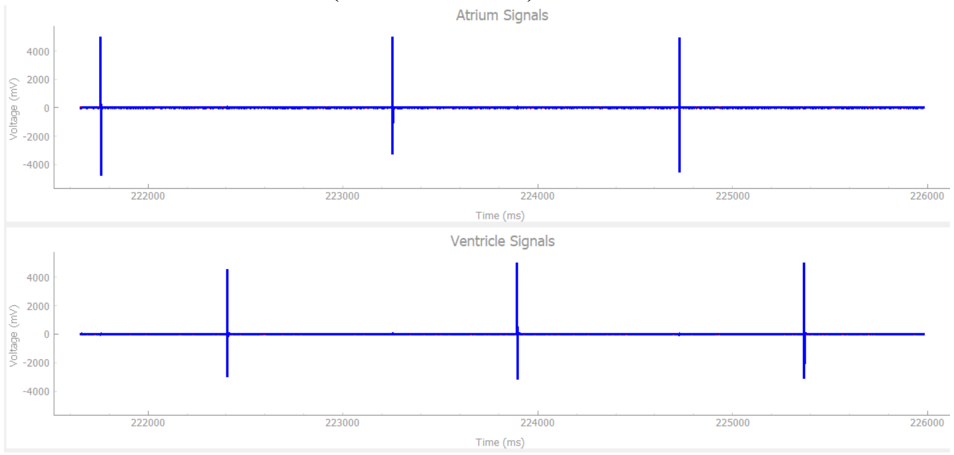
AAIR Mode	
Parameters	**HeartView Atrium Pacing**
Low Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 5 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)  Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards, while inhibiting a pulse to the heart when a natural heart beat is sensed. **Result: PASS**

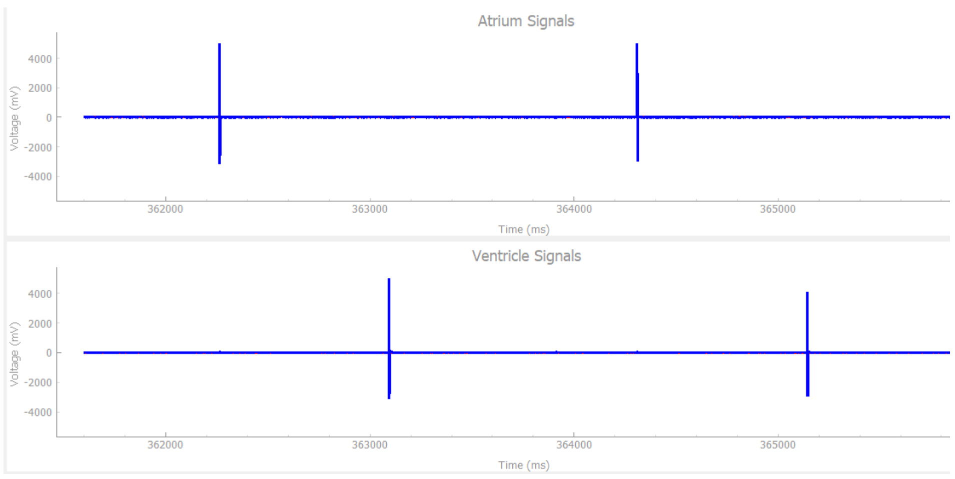
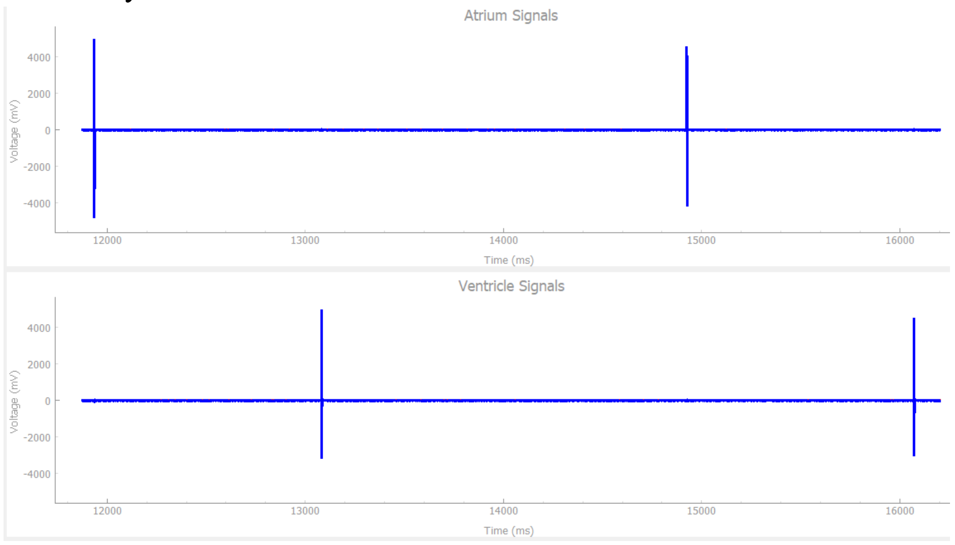
High Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 30 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)  Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards, while inhibiting a pulse to the heart when a natural heart beat is sensed. **Result: PASS**

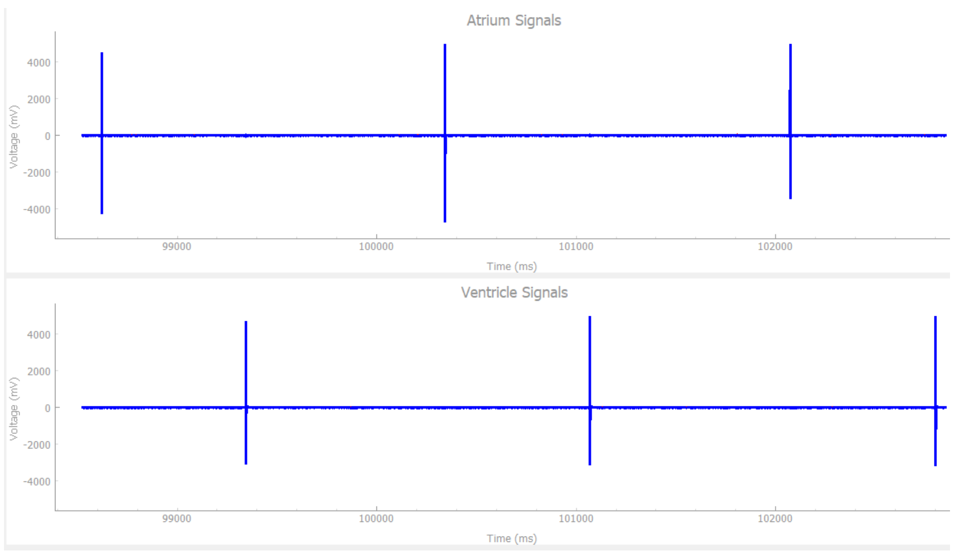
VVIR Mode	
Parameters	**HeartView Atrium Pacing**
Low Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 5 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)

	 After motion is stopped (Recovery Time)  Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards, while inhibiting a pulse to the heart when a natural heart beat is sensed. **Result: PASS**
High Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 30 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)  Above we see that the heart beats faster while motion is detected and the

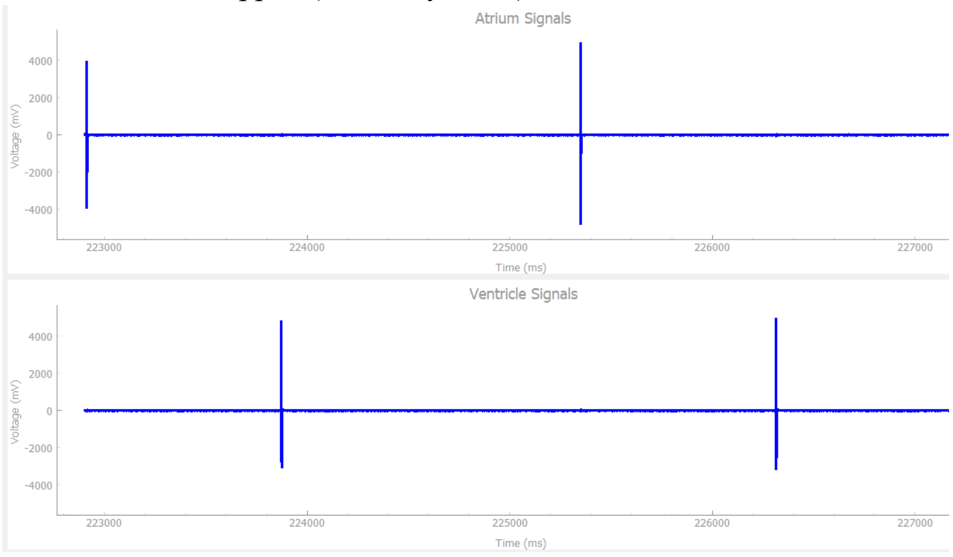
	recovery happens in a timely manner afterwards, while inhibiting a pulse to the heart when a natural heart beat is sensed. **Result: PASS**

DOOR Mode	
Parameters	**HeartView Atrium Pacing**
Low Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 5 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)  After motion is stopped (Recovery Time)

	 Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards. **Result: PASS**
High Threshold Value Rate Adaptive Pulse Amplitude: 5V Pulse Width: 1ms Activity Threshold: 30 Response Factor: 8	Before any movement  While motion is detected (Reaction Time)



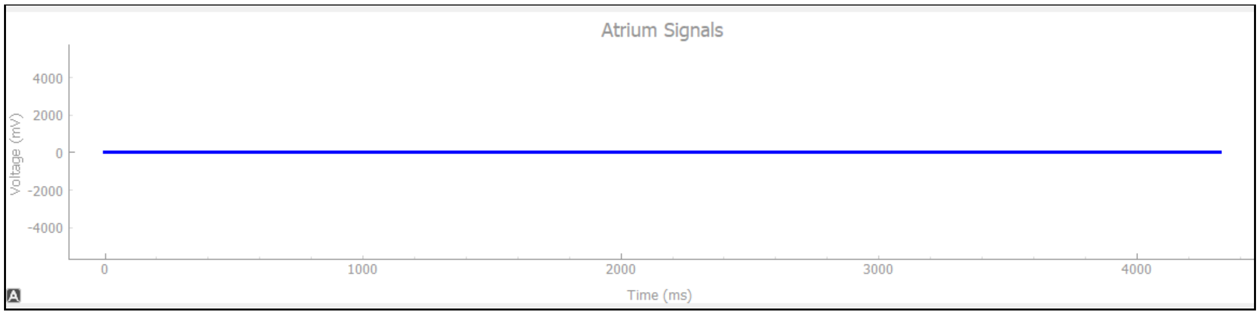
After motion is stopped (Recovery Time)



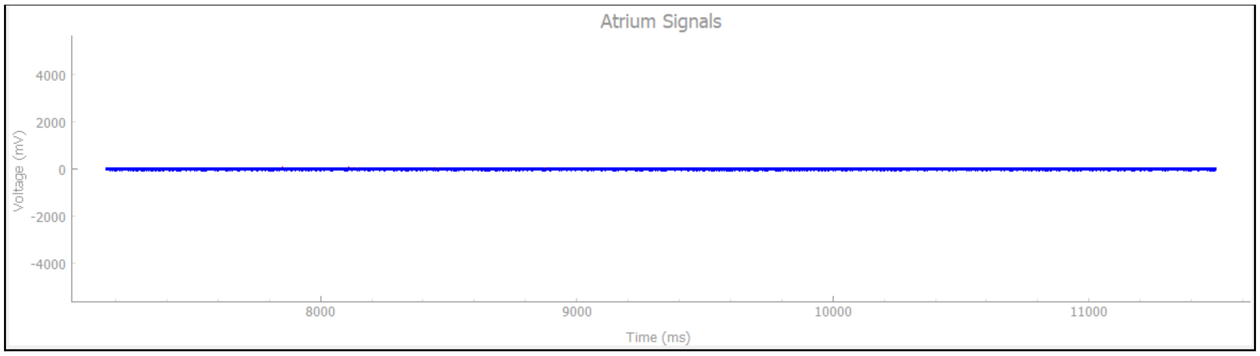
Above we see that the heart beats faster while motion is detected and the recovery happens in a timely manner afterwards. **Result: PASS**

Unusual Error while testing AAIR

When we changed our mode to AAIR, our Heartview output was a noisy signal, however, after the board was unplugged and flashed again, the mode worked as intended. This error is documented because during the lab demo, we faced the same issue, however, we were unsure of what to do, making it seem like we failed to implement AAIR logic correctly



Regular Atrium plot in Heartview prior to pacing (AAIR)



Noisy Atrium plot after changing mode to AAIR

Part 2 - DCM Design

Current Requirements

Parameter	Programmable Values	Increment	Nominal	Tolerance
A or V Pulse Amplitude Regulated	Off, 0.1-5.0V	0.1V	5 V	+/- 12 ms
A or V Pulse Width	1-30 ms	1 ms	1 ms	1 ms
A or V Sensitivity	0-5V	0.1V	-	+/- 2%

Table 3: New Requirements

1. Serial communication is implemented to receive data and transmit between the DCM and pacemaker

2. System is able to set, store, transmit the programmable parameter data, and verify it is stored correctly on the device
3. The parameters, programmable values, increment, nominal and tolerance are all generated with the requirements from the table given to implement for all the modes AOO, VOO, AAI, VVI, DOO, AOOR, VOOR, AAIR, VVIR and DOOR

Requirement Changes That are Likely

1. If we wanted to extend the modes we could implement a DDDR mode
2. Only inhibit ventricular pacing when holding down the pushbutton keeping the atrial functionality unaffected
3. Ventricular pacing is enabled once the pushbutton is released

Design Decisions That are Likely to Change

1. The capability of users that can be stored in the device can be altered so it can handle more users
2. When creating the new extended DDR mode we would implement an AV delay
3. The ventricular pacing will be dependent on the pushbutton

External Libraries

Package	Usage
tkinter	Build GUI design
numpy	Data manipulation
Pyserial	Serial Communication
struct	Byte pack/unpack for serial communication
matplotlib	Plot Atrial and Ventricular activity

WriteListOfValues Function

```
def writeListOfValues(values):
   pattern ="<BBBBBfBfBBHHffHBBBHBBBB"
   assert (len(pattern[1:]) == len(values)), "Pattern and values arrays are not
equal in size"
   endian = pattern[0]
   pattern = pattern[1:]
   packet = []
```

```
    for i in range(0, len(values)):
        packedData = pack((endian + pattern[i]), values[i])
        packet.append(packedData)
    for pac in packet:
        ser.write(pac)
```

Purpose
This Function is used to set parameters to the Pacemaker from the DCM via serial communication
Arguments
values : list/array
Explanation
When this function is called, the DCM sets the parameters passed in the values list to the pacemaker. It packs the variables according to the stream pattern outlined in the Serial Communication Protocol section of this document.
Return Type: void

SerialRead Function

```
def serialRead():
    pattern =
"<BBBfBfBBhhffhBBBhBBBBffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff"
    res = ser.read(278)
    res = list(unpack(pattern, res))
    return res
```

Purpose
This Function is used to read parameters from the Pacemaker to DCM via serial communication
Explanation
Using the data stream pattern outlined in the Serial Communication Protocol section of this document, this function reads the data being sent from the Pacemaker to the DCM and unpacks it.
Return Type: list

GetEgramData Function

SerialRead Function

```
def getEgramData():
    count=0
    def egram(i):
        def signalProcess(n):
            return (n-0.5)*3.3
        global count
        global yA
        global yV
        params = [16 , 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0, 0 , 0]
        writeListOfValues(params)
        y = serialRead()
        yA = yA[30:]+list(map(signalProcess,y[21:51]))
        yV = yV[30:]+list(map(signalProcess,y[51:]))
        count += 1
        x = [(count+j) for j in range(150)]
        axs[0].cla()
        axs[1].cla()
        axs[0].set_yticks([-4,-3.5,-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0])
        axs[0].set_ylim(-4, 4)
        axs[1].set_yticks([-4,-3.5,-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0])
        axs[1].set_ylim(-4, 4)
        axs[0].grid()
        axs[1].grid()
        axs[0].set_ylabel("Amplitude(V)")
        axs[1].set_xlabel("Time(msec)")
        axs[1].set_ylabel("Amplitude(V)")
        axs[0].plot(x, yA, c='r', label='Artial',linewidth=3.0)
        axs[1].plot(x, yV, c='b', label='Ventricular',linewidth=3.0)
        axs[0].set_title('Atrial')
        axs[1].set_title('Ventricular')
    fig, axs = plt.subplots(2)
    animate = animation.FuncAnimation(fig, egram, interval=150)
    plt.show()
```

Purpose
This Function is used to read atrial and ventricular sample data from the Pacemaker and display an electrocardiogram on the DCM.
Explanation
Using the matplotlib library, a real time plot of the atrial and ventricular activity is plotted using a functional animation. Initially all sample data is set to 0, and as data is sent from the pacemaker to the dcm, the plot is updated to display the new data. Every 150 msec a new sample is sent and displayed on the window.
Return Type: void

DCMapp Class

```
class DCMapp(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self._frame = None
        self.switch_frame(WelcomePage)
        self.geometry("400x700")

    def switch_frame(self, frame_class):
        new_frame = frame_class(self)
        if self._frame is not None:
            self._frame.destroy()
        self._frame = new_frame
        self._frame.pack()
```

Purpose
This class acts as a general frame which helps with the switch from 1 frame/screen to another. Using the destroy function we can take away the current screen and show a new frame based on our choice.
Secret of Class
There is no secret for this class in terms of data processing.
Public Functions In the Class
We are using two functions in this class, the first one is the constructor __init__(self) function with self as the parameter, and the switch_frame function with self and frame_class as parameters. The __init__ function is used to set the initial welcome page and the dimensions for future current and future screens. The welcome page screen is defined in its own class. The switch_frame(self, frame_class) function is used to display a new screen of choice by removing the current screen and having it replaced by the new one.
How It Works
switch_frame(self, frame_class)
We simply take an input into the switch_frame function which takes another frame class as an input. We have the frame details defined in individual classes so when it is sent into the swtich_frame function as a parameter, the new frame is outputted and the old frame is destroyed.

__init__(self)
The parameter self represents the current frame we are on which is the welcome screen. We set the margins for the screen as well as call the first screen which is what is outputted.

Private functions and global variables were not used in this class

WelcomePage Class

```
class WelcomePage(tk.Frame):
   def __init__(self, master):

       tk.Frame.__init__(self, master)

       headerFont = ("Times", 23, "bold")
       welcomelabel = tk.Label(self, text="PACEMAKER DCM", font = headerFont, pady = 45)
       welcomelabel.pack()

       signInButton = tk.Button (self,text="Sign In", command=lambda: master.switch_frame(SignInPage))
       signInButton.pack()

       registerButton = tk.Button (self,text="Register", command=lambda: master.switch_frame(RegisterPage))
       registerButton.pack()
```

Purpose
This class is used to initialize the welcome page which is the first screen the user will see once the software is loaded. The welcome page contains two options for users which is the sign in option and the register option.
Secret of Class
There is no secret for this class in terms of data processing.

Public Functions In the Class
We are only using one function in this class which is the constructor __init__(self, master). This function takes in two parameters which are self and master. Self refers to the current class we are working on (WelcomePage) and master refers to the current screen that is seen by the viewer. We are setting the welcome page in this function.
How It Works
__init__(self, master)
As explained above we take two inputs into the function. In the function itself we are firstly setting the font and the text size of the welcome screen text. There are two options on the welcome screen, we can either go to the sign in screen or the register page. In order to give access to complete these operations two buttons are created in the function to navigate between the two screens. We see that switch_frame function is called again to switch frames on the press of the button depending on which button is pressed.

Private functions and global variables were not used in this class

SignInPage Class

```python
class SignInPage(tk.Frame):
   def __init__(self, master):

       tk.Frame.__init__(self, master)
       headerFont = ("Times", 16, "bold")
       signInHeader = tk.Label(self, text="Sign In Page", font = headerFont, pady= 20)
       signInHeader.pack()

       usernameLabel = tk.Label(self, text="Username").pack()
       userInput = tk.Entry(self, width=20)
       userInput.focus()
       userInput.pack()

       passwordLabel = tk.Label(self, text="Password").pack()
       passInput = tk.Entry(self, show="*", width=20)
       passInput.pack()

       def handleSignIn():
           f = open('users/masterlist.txt','r')
           userList=f.read().split(',')
           f.close()
           usernameIn=userInput.get()
           passwordIn=passInput.get()

           password=""
           if(usernameIn in userList):
               f = open(f"users/{usernameIn}.txt",'r')
               contents=f.read().split("*")
               f.close()

               credentials = contents[0].split('\n')
               password = credentials[1]
```

```
                f = open(f"users/CurrUser.txt", "w")
                f.write(usernameIn)
                f.close()

                if(password == passwordIn):
                    master.switch_frame(HomePage)
                else:
                    messagebox.showwarning('Warning', 'Invalid Credentials')
            else:
                messagebox.showwarning('Warning', 'User Not Found')

        SignInButton = tk.Button (self, text="Sign In", command=handleSignIn)
        SignInButton.pack()

        backButton = tk.Button (self, text="Back", command=lambda: master.switch_frame(WelcomePage))
        backButton.pack()
```

Purpose
This class is used to initialize the sign in page where users will be able to login using a username and a password. Only if there is an existing account in the system will the user be allowed to login. There are two separate parts to this method.
Secret of Class
In this method we are using text files to retrieve client information. The users masterlist file is used to store all the usernames and passwords of the individuals that have previously registered. We also use a second text file called CurrUser to store the username of the current user who has logged in so we use this to retrieve their previous stored information.
Public Functions In the Class
We are using two functions in this class which are the constructor __init__(self, master). This function takes in two parameters which are self and master. Self refers to the current class we are working on (Sign In Page) and master refers to the current screen that is seen by the viewer. HandleSignIn() has no parameters and is used to check whether the user account exists.
How It Works
__init__(self, master)
As explained above we take two inputs into the function. In the function itself we are firstly setting the font and the text size of the sign in screen. Then we are setting two entry boxes for the user to type in their username and password. We also implement a back button which takes us back to the welcome screen.

handleSignIn()
The second function, handleSignIn, is inside __init__ and it opens the master list text file, reads the inputs, and checks to see if the user exists. If the user does not exist an error is thrown. We then check if the password is correct, if it isn't an error is thrown. If the password is correct we use the switch_frame function to switch to the home page.
Private functions and global variables were not used in this class

RegisterPage Class

```
class RegisterPage(tk.Frame):
   def __init__(self, master):
       tk.Frame.__init__(self, master)
       headerFont = ("Times", 16, "bold")
       RegisterHeader = tk.Label(self, text="Registration Page", font = headerFont, pady= 5)
       RegisterHeader.pack()
       header2Font = ("Times", 13, "bold")
       RegisterSubHeader = tk.Label(self, text="Enter credentials to create account", font = header2Font,
pady= 10)
       RegisterSubHeader.pack()

       usernameLabel = tk.Label(self, text="Username").pack()
       userInput = tk.Entry(self, width=20)
       userInput.focus()
       userInput.pack()

       passwordLabel = tk.Label(self, text="Password").pack()
       passInput = tk.Entry(self, width=20)
       passInput.pack()

       def handleRegister():
           f = open('users/masterlist.txt','r')
           userList=f.read().split(',')
           f.close()
           if(len(userList) >= 10):
               messagebox.showwarning('Error', 'System user limit reached')
               return
           else:
               usernameIn=userInput.get()
               passwordIn=passInput.get()

               if(usernameIn in userList):
                   messagebox.askretrycancel('retry', 'User already exists!')
               else:
                   if (" " in usernameIn) or (" " in passwordIn) or (not usernameIn) or (not passwordIn):
                       messagebox.askretrycancel('retry', 'Invalid username or password')
                   else:
                       userList.append(usernameIn)
                       updatedList=",".join(userList)

                       #Add user to master list
                       f = open("users/masterlist.txt","w")
                       f.write(updatedList)
                       f.close()
                       #Get default data
                       f = open("users/DefaultUser.txt","r")
                       defaultInfo=f.read()
                       f.close()
                       #Add new user
                       f = open(f"users/{usernameIn}.txt",'w')
                       f.write(usernameIn+"\n"+passwordIn+"\n")
                       f.write(defaultInfo)
                       f.close()
                       #update current user
```

```
                    f = open(f"users/CurrUser.txt", "w")
                    f.write(usernameIn)
                    f.close()
                    master.switch_frame(HomePage)
        RegisterButton = tk.Button (self, text="Register", command=handleRegister)
        RegisterButton.pack()
        backButton = tk.Button (self, text="Back", command=lambda: master.switch_frame(WelcomePage))
        backButton.pack()
```

Purpose
This class is used to initialize the register page where users will be able to register using a username and a password. If there is an existing account in the system with the username that is imputed then an error is thrown. There are two parts to this class.
Secret of Class
In this method we are using text files to store client information. The users masterlist file is used to store all the usernames and passwords of the individuals that have previously registered. So we are writing to this file with the new user info. We also use a second text file called CurrUser to store the username of the current user who has logged in. In this case the current registered user. We have a third text file DefaultUser which stores default programmable parameters for a new user when they register. Finally we create a new text file for the user and store their username, password, and default programmable parameters.
Public Functions In the Class
We are using two functions in this class which are the constructor __init__(self, master). This function takes in two parameters which are self and master. Self refers to the current class we are working on (Register page) and master refers to the current screen that is seen by the viewer. HandleRegister() has no parameters and is used to create a new user if the username doesn't already exist.
How It Works
__init__(self,master)
As explained above we take two inputs into the function. In the function itself we are firstly setting the font and the text size of the register screen text. Then we are setting two entry boxes for the user to type in their username and password. We also implement a back button which takes us back to the welcome screen.

handleRegister()
The second function, handleRegister, is inside __init__ and it opens the master list text file, reads the inputs, and checks to see if the user exists. If the user exists an error is thrown since two users cannot have the same username. We then take the password and store the username, password, and default parameters in a new text file which acts as the user account. After the text file is made we use switch_frame function to go to the home page. There is also error checking implemented which means that new users cannot have a blank username or password.

Private functions and global variables were not used in this class

HomePage Class

```
class HomePage(tk.Frame):
    def __init__(self, master):
        tk.Frame.__init__(self, master)

        headerFont = ("Times", 16, "bold")
        HomeHeader = tk.Label(self, text="Home Page:", font = headerFont, pady= 5)
        HomeHeader.grid(row=1,column=0)

        connect = tk.Label(self, text='Connected', foreground='green')
        diffpacemaker = tk.Label(self, text='Different Pacemaker Detected', foreground='red')
        setpacemaker = tk.Label(self, text='Current Pacemaker set to User', foreground='green')

        #getting the pacemaker ID for the user
        f = open("users/CurrUser.txt","r")
        user=f.read()
        f.close()

        f = open(f"users/{user}.txt",'r')
        contents=f.read().split("*")
        f.close()
        credentials = contents[0].split('\n')
        username = credentials[0]
        password = credentials[1]
        AOO = contents[1].split('\n')[2:-1]
        AAI = contents[2].split('\n')[2:-1]
        VOO = contents[3].split('\n')[2:-1]
        VVI = contents[4].split('\n')[2:-1]
        DOO = contents[5].split('\n')[2:-1]
        AOOR = contents[6].split('\n')[2:-1]
        VOOR = contents[7].split('\n')[2:-1]
        AAIR = contents[8].split('\n')[2:-1]
        VVIR = contents[9].split('\n')[2:-1]
        DOOR = contents[10].split('\n')[2:-1]
        ID = int(contents[11])

        def updateFile():
            f = open(f"users/{username}.txt", "w")
            f.write(username+"\n"+password+"\n")
            f.write("*\nAOO\n")
            for i in AOO:
                f.write(i+"\n")
            f.write("*\nAAI\n")
            for i in AAI:
                f.write(i+"\n")
            f.write("*\nVOO\n")
            for i in VOO:
                f.write(i+"\n")
            f.write("*\nVVI\n")
            for i in VVI:
                f.write(i+"\n")
            f.write("*\nDOO\n")
            for i in DOO:
                f.write(i+"\n")
            f.write("*\nAOOR\n")
            for i in AOOR:
                f.write(i+"\n")
            f.write("*\nVOOR\n")
            for i in VOOR:
                f.write(i+"\n")
            f.write("*\nAAIR\n")
            for i in AAIR:
                f.write(i+"\n")
            f.write("*\nVVIR\n")
            for i in VVIR:
                f.write(i+"\n")
            f.write("*\nDOOR\n")
            for i in DOOR:
                f.write(i+"\n")
            f.write("*\n")
            f.write(ID+"\n")
            f.close()

        currpacemaker = ID
        params = [16 , 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0, 0 , 0]
        writeListOfValues(params)
        newpacemaker = serialRead()[0]

        #if 0, no pacemaker user before, therefore set the current pacemaker as the user's pacemaker
        if currpacemaker == 0:
            currpacemaker = newpacemaker
            ID = str(currpacemaker)
            updateFile()
            pacediff = "false"
            paceset = "true"

        elif currpacemaker != 0:
            paceset = "false"
            if currpacemaker == newpacemaker:
                pacediff = "false"

            else:
                pacediff = "true"

        #once the pacemaker will actually be connected of not, this function will be used to check for that
connection
        def connection():
            connect.forget()

            connect.grid(row=3,column=0)
            #comparing pacemaker id to see if its user's pacemaker
            if (pacediff == "true"):
                diffpacemaker.grid(row=4,column=0)
            if paceset == "true":
                setpacemaker.grid(row=4,column=0)

        ConnectedButton = tk.Button (self, text="Check Pacemaker Connection", command=connection)
        ConnectedButton.grid(row=2,column=0)


        header2Font = ("Times", 13, "bold")
        HomeSubHeader = tk.Label(self, text="Select mode you want to view for your pacemaker:", font =
 header2Font, pady= 10)
        HomeSubHeader.grid(row=5,column=0)

        AOOButton = tk.Button (self, text="AOO Mode", command=lambda: master.switch_frame(AOOPage))
        AOOButton.grid(row=6,column=0)

        VOOButton = tk.Button (self, text="VOO Mode", command=lambda: master.switch_frame(VOOPage))
        VOOButton.grid(row=7,column=0)

        AAIButton = tk.Button (self, text="AAI Mode", command=lambda: master.switch_frame(AAIPage))
        AAIButton.grid(row=8,column=0)

        VVIButton = tk.Button (self, text="VVI Mode", command=lambda: master.switch_frame(VVIPage))
        VVIButton.grid(row=9,column=0)

        DOOButton = tk.Button (self, text="DOO Mode", command=lambda: master.switch_frame(DOOPage))
        DOOButton.grid(row=10,column=0)

        AOORButton = tk.Button (self, text="AOOR Mode", command=lambda: master.switch_frame(AOORPage))
        AOORButton.grid(row=11,column=0)

        VOORButton = tk.Button (self, text="VOOR Mode", command=lambda: master.switch_frame(VOORPage))
        VOORButton.grid(row=12,column=0)

        AAIRButton = tk.Button (self, text="AAIR Mode", command=lambda: master.switch_frame(AAIRPage))
        AAIRButton.grid(row=13,column=0)

        VVIRButton = tk.Button (self, text="VVIR Mode", command=lambda: master.switch_frame(VVIRPage))
        VVIRButton.grid(row=14,column=0)

        DOORButton = tk.Button (self, text="DOOR Mode", command=lambda: master.switch_frame(DOORPage))
        DOORButton.grid(row=15,column=0)

        EGRAMButton = tk.Button (self, text="Display egram", command=getEgramData)
        EGRAMButton.grid(row=16,column=0)

        def signOut():
            f = open("users/CurrUser.txt","w")
            user=f.write("")
            f.close()
            master.switch_frame(WelcomePage)
        signOutButton = tk.Button (self, text="Sign Out", command=signOut )
        signOutButton.grid(row=17,column=0)
```

Purpose
This class is used to initialize the home page where users will be able to click buttons to check all of their pacing modes. There are 4 buttons each representing a pacing mode. When pressed, it takes you to a new screen where you can edit specific programmable parameters related to that pacing mode.
Secret of Class
There is no particular secret to this function.
Public Functions In the Class
We are using three functions in this class which are the constructor __init__(self,master). This function

takes in two parameters which are self and master. Self refers to the current class we are working on (home page) and master refers to the current screen that is seen by the viewer. Connection() has no parameters and is used to create a link with the pacemaker, and signOut() which also has no parameters and allows you to go back to the welcome page.
How It Works
__init__(self,master)
As explained above we take two inputs into the function. In the function itself we are firstly setting the font and the text size of the home page text. Then we are setting four buttons each representing a pacing mode. When a button is pressed the screen with the programmable parameters of that mode will show up using the switch_frame function.

updateFile()
This function is used to write all the saved changes to the text file containing the user information. As mentioned before we store user information in textfiles. As soon as the user hits the save button this function is called on where we read the saved values and write them to the text file replacing any old information on it.

connection()
This function is used to check whether the pacemaker is connected or not once the serial communication is established. Before this function, there are a set of conditional statements which run in order to determine what pacemaker is connected to the DCM and whether that pacemaker is the same as the one the user uses. In order to do this, a unique ID is hardcoded into each pacemaker device, which will be retrieved from serial communication. If the user is new, the ID of their pacemaker is set to 0 initially, however, when they first connect to the new pacemaker, the ID of the pacemaker will be updated in their file. If the user already has a non zero unique ID, the DCM will check if the ID of the pacemaker is the same as the ID in the file of the user. If it is, then it will display connected in the DCM front end. If it is a different pacemaker, it will display that a different pacemaker was detected.

signOut()
This function creates a sign out button and on press we use the switch_frame function to go back to the welcome page screen. We also update the text file CurrUser, which stores the current user. We write a blank onto the text file to indicate that there is no account that is currently pulled up.
Private functions and global variables were not used in this class

AOOPage Class

```
class AOOPage(tk.Frame):

   def __init__(self, master):

       #getting the pacemaker ID for the user
       f = open("users/CurrUser.txt","r")
       user=f.read()
       f.close()

       f = open(f"users/{user}.txt",'r')
       contents=f.read().split("*")
       f.close()
       credentials = contents[0].split('\n')
       username = credentials[0]
```

```
        password = credentials[1]
        AOO = contents[1].split('\n')[2:-1]
        AAI = contents[2].split('\n')[2:-1]
        VOO = contents[3].split('\n')[2:-1]
        VVI = contents[4].split('\n')[2:-1]
        DOO = contents[5].split('\n')[2:-1]
        AOOR = contents[6].split('\n')[2:-1]
        VOOR = contents[7].split('\n')[2:-1]
        AAIR = contents[8].split('\n')[2:-1]
        VVIR = contents[9].split('\n')[2:-1]
        DOOR = contents[10].split('\n')[2:-1]
        ID = int(contents[11])

        def updateFile():
            f = open(f"users/{username}.txt", "w")
            f.write(username+"\n"+password+"\n")
            f.write("*\nAOO\n")
            for i in AOO:
                f.write(i+"\n")
            f.write("*\nAAI\n")
            for i in AAI:
                f.write(i+"\n")
            f.write("*\nVOO\n")
            for i in VOO:
                f.write(i+"\n")
            f.write("*\nVVI\n")
            for i in VVI:
                f.write(i+"\n")
            f.write("*\nDOO\n")
            for i in DOO:
                f.write(i+"\n")
            f.write("*\nAOOR\n")
            for i in AOOR:
                f.write(i+"\n")
            f.write("*\nVOOR\n")
            for i in VOOR:
                f.write(i+"\n")
            f.write("*\nAAIR\n")
            for i in AAIR:
                f.write(i+"\n")
            f.write("*\nVVIR\n")
            for i in VVIR:
                f.write(i+"\n")
            f.write("*\nDOOR\n")
            for i in DOOR:
                f.write(i+"\n")
            f.write("*\n")
            f.write(str(ID)+"\n")
            f.close()

        def sendAOOData():
            params = [16, 34, 0, 1, 60, 3.5, 1, 3.5, 1, 120, 250, 320, 0.001, 0.0025, 250, 0, 0, 120, 150,
15, 30, 8, 5]
```

```
            params[4] = int(AOO[0])
            params[9] = int(AOO[1])
            try:
                params[7] = float(AOO[2])
            except:
                params[7] = 0.0
            try:
                params[8] = int(AOO[3])
            except:
                params[8] = 0
            writeListOfValues(params)

        #change mode and send previous values
        sendAOOData()

        def saveChanges():

            try:
                LRL=int(valLowerRateLimit.get())
                URL=int(valUpperRateLimit.get())
            except:
                messagebox.showwarning('Warning', 'Invalid Input')
            try:
                AA=float(valAtrialAmplitude.get())
            except:
                AA=valAtrialAmplitude.get()
                if(AA!="OFF"):
                    messagebox.showwarning('Warning', 'Invalid Input')
            try:
                APW=int(valAtrialPulseWidth.get())
            except:
                APW=valAtrialPulseWidth.get()
                if(APW!="OFF"):
                    messagebox.showwarning('Warning', 'Invalid Input')

            LRL_Range1 = (LRL % 5 == 0) and (LRL >= 30) and (LRL < 50)
            LRL_Range2 = (LRL % 1 == 0) and (LRL >= 50) and (LRL < 90)
            LRL_Range3 = (LRL % 5 == 0) and (LRL >= 90) and (LRL <= 175)
            checkLowerRateLimit = (LRL >= 30) and (LRL <= 175) and (LRL_Range1 or LRL_Range2 or LRL_Range3)
and (LRL <= URL)
            checkUpperRateLimit = (URL % 5 == 0) and (URL >= 50) and (URL <= 175)
            checkAtrialAmplitude = not(isinstance(AA, str)) and (((AA*10) % 1 == 0) and ((AA >= 0.1) and (AA
<= 5.0))) or (AA == "OFF")
            checkAtrialPulseWidth =  (not(isinstance(APW, str)) and (((APW) % 1 == 0) and (APW >= 1) and (APW
<= 30))) or (APW == "OFF")

            if(checkLowerRateLimit and checkUpperRateLimit and checkAtrialAmplitude and
checkAtrialPulseWidth):
                AOO[0] = valLowerRateLimit.get()
                AOO[1] = valUpperRateLimit.get()
                AOO[2] = valAtrialAmplitude.get()
                AOO[3] = valAtrialPulseWidth.get()
                sendAOOData()
                updateFile()
                master.switch_frame(HomePage)
            else:
                messagebox.showwarning('Warning', 'Invalid Input')

        tk.Frame.__init__(self, master)

        BackButton = tk.Button (self, text="Back", width = 13, command=lambda: master.switch_frame(HomePage))
        BackButton.grid(row=0,column=0)

        SaveButton = tk.Button (self, text="Save", width = 13, command=saveChanges)
        SaveButton.grid(row=0,column=1)

        headerFont = ("Times", 16, "bold")
        Header1 = tk.Label(self, text="Mode:", font = headerFont, pady= 5)
        Header1.grid(row=1,column=0)
        Header2 = tk.Label(self, text="AOO", font = headerFont, pady= 5)
        Header2.grid(row=1,column=1)

        Font3 = ("Times", 11, "bold")

        LowerRateLimit = tk.Label(self, text="Lower Rate Limit (ppm)", font = Font3).grid(row=3,column=0)
        textLRL = tk.StringVar()
        textLRL.set(AOO[0])
        valLowerRateLimit = tk.Entry(self, textvariable=textLRL, width=13)
        valLowerRateLimit.focus()
        valLowerRateLimit.grid(row=3,column=1)

        UpperRateLimit = tk.Label(self, text="Upper Rate Limit (ppm)", font = Font3).grid(row=4,column=0)
        textURL = tk.StringVar()
        textURL.set(AOO[1])
        valUpperRateLimit = tk.Entry(self, textvariable=textURL, width=13)
        valUpperRateLimit.grid(row=4,column=1)

        MaximumSensorRate = tk.Label(self, text="Maximum Sensor Rate (ppm)", font =
Font3).grid(row=5,column=0)
        valMaximumSensorRate = tk.Label(self, text="120", font = Font3).grid(row=5,column=1)
        FixedAVDelay = tk.Label(self, text="Fixed AV Delay (ms)", font = Font3).grid(row=6,column=0)
        valFixedAVDelay = tk.Label(self, text="150", font = Font3).grid(row=6,column=1)
        DynamicAVDelay = tk.Label(self, text="Dynamic AV Delay", font = Font3).grid(row=7,column=0)
        valDynamicAVDelay = tk.Label(self, text="OFF", font = Font3).grid(row=7,column=1)
        MinDynAVDelay = tk.Label(self, text="Minimum Dynamic AV Delay (ms)", font =
Font3).grid(row=8,column=0)
        valMinDynAVDelay = tk.Label(self, text="50", font = Font3).grid(row=8,column=1)
        SenAVDelayOffset = tk.Label(self, text="Sensed AV Delay Offset (ms)", font =
Font3).grid(row=9,column=0)
        valSenAVDelayOffset = tk.Label(self, text="OFF", font = Font3).grid(row=9,column=1)

        AtrialAmplitude = tk.Label(self, text="Atrial Amplitude (V)", font = Font3).grid(row=10,column=0)
        textAA = tk.StringVar()
        textAA.set(AOO[2])
        valAtrialAmplitude = tk.Entry(self, textvariable=textAA, width=13)
        valAtrialAmplitude.grid(row=10,column=1)
        VentricularAmplitude = tk.Label(self, text="Ventricular Amplitude (V)", font =
Font3).grid(row=11,column=0)
        valVentricularAmplitude = tk.Label(self, text="3.5", font = Font3).grid(row=11,column=1)

        AtrialPulseWidth = tk.Label(self, text="Atrial Pulse Width (ms)", font = Font3).grid(row=12,column=0)
        textAPW = tk.StringVar()
        textAPW.set(AOO[3])
        valAtrialPulseWidth = tk.Entry(self, textvariable=textAPW, width=13)
        valAtrialPulseWidth.grid(row=12,column=1)

        VentricularPulseWidth = tk.Label(self, text="Ventricular Pulse Width (ms)", font =
Font3).grid(row=13,column=0)
        valVentricularPulseWidth = tk.Label(self, text="0.4", font = Font3).grid(row=13,column=1)
        AtrialSensitivity = tk.Label(self, text="Atrial Sensitivity (V)", font = Font3).grid(row=14,column=0)
        valAtrialSensitivity = tk.Label(self, text="0.75", font = Font3).grid(row=14,column=1)
        VentricularSensitivity = tk.Label(self, text="Ventricular Sensitivity (V)", font =
Font3).grid(row=15,column=0)
        valVentricularSensitivity = tk.Label(self, text="2.5", font = Font3).grid(row=15,column=1)
        VRP = tk.Label(self, text="VRP (ms)", font = Font3).grid(row=16,column=0)
        valVRP = tk.Label(self, text="320", font = Font3).grid(row=16,column=1)
        ARP = tk.Label(self, text="ARP (ms)", font = Font3).grid(row=17,column=0)
        valARP = tk.Label(self, text="250", font = Font3).grid(row=17,column=1)
        PVARP = tk.Label(self, text="PVARP (ms)", font = Font3).grid(row=18,column=0)
        valPVARP = tk.Label(self, text="250", font = Font3).grid(row=18,column=1)
        PVARPExtension = tk.Label(self, text="PVARP Extension (ms)", font = Font3).grid(row=19,column=0)
        valPVARPExtension = tk.Label(self, text="OFF", font = Font3).grid(row=19,column=1)
        Hysteresis = tk.Label(self, text="Hysteresis (ppm)", font = Font3).grid(row=20,column=0)
        valHysteresis = tk.Label(self, text="OFF", font = Font3).grid(row=20,column=1)
        RateSmoothing = tk.Label(self, text="Rate Smoothing (%)", font = Font3).grid(row=21,column=0)
        valRateSmoothing = tk.Label(self, text="OFF", font = Font3).grid(row=21,column=1)
        ATRDuration = tk.Label(self, text="ATR Duration (cc)", font = Font3).grid(row=22,column=0)
        valATRDuration = tk.Label(self, text="20", font = Font3).grid(row=22,column=1)
        ATRFallbackMode = tk.Label(self, text="ATR Fallback Mode", font = Font3).grid(row=23,column=0)
        valATRFallbackMode = tk.Label(self, text="OFF", font = Font3).grid(row=23,column=1)
        ATRFallbackTime = tk.Label(self, text="ATR Fallback Time (min)", font = Font3).grid(row=24,column=0)
        valATRFallbackTime = tk.Label(self, text="30", font = Font3).grid(row=24,column=1)
        ActivityThreshold = tk.Label(self, text="Activity Threshold", font = Font3).grid(row=25,column=0)
        valActivityThreshold = tk.Label(self, text="MED", font = Font3).grid(row=25,column=1)
        ReactionTime = tk.Label(self, text="Reaction Time (sec)", font = Font3).grid(row=26,column=0)
        valReactionTime = tk.Label(self, text="30", font = Font3).grid(row=26,column=1)
        ResponseFactor = tk.Label(self, text="Response Factor", font = Font3).grid(row=27,column=0)
        valResponseFactor = tk.Label(self, text="8", font = Font3).grid(row=27,column=1)
        RecoveryTime = tk.Label(self, text="Recovery Time (min)", font = Font3).grid(row=28,column=0)
        valRecoveryTime = tk.Label(self, text="5", font = Font3).grid(row=28,column=1)
```

Purpose
The purpose of this class is to be able to see the programmable parameters for the AOO pacing mode and edit them according to specific ranges of values. Once the user has changed the programmable parameters they will be able save them if and only if they increase or decrease the parameters in the specific ranges that are set for each programmable parameter. Changes are saved if inputs are valid.

Secret of Class

According to our programmable parameter documents for the assignment we were given specific ranges for which parameters can increase or decrease by. We implemented conditions to check whether the user entered correct parameters when changing these values which is the whole secret behind the class.

Public Functions In the Class
We are using three functions in this class. The constructor __init__(self,master). This function takes in two parameters which are self and master. Self refers to the current class we are working on (AOO page) and master refers to the current screen that is seen by the viewer. We are using updateFile() which is a function that has no parameters and updates the user text file to update the new parameters. Finally we have saveChanges() which does not have any parameters and checks if the entered parameter changes are valid to save.

How It Works
__init__(self,master)
As explained above we take two inputs into the function. In the function itself we are firstly setting the font and the text size of the AOO page text. This function also sets up text boxes for each possible programmable parameter, however there are only some parameters that can be edited in this pacing mode and therefore those are set as editable and the others are greyed out.

updateFile()
This function is used to write all the saved changes to the text file containing the user information. As mentioned before we store user information in textfiles. As soon as the user hits the save button this function is called on where we read the saved values and write them to the text file replacing any old information on it.

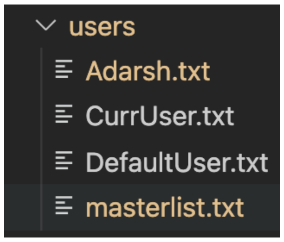
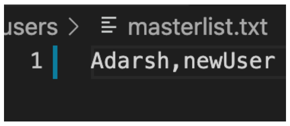
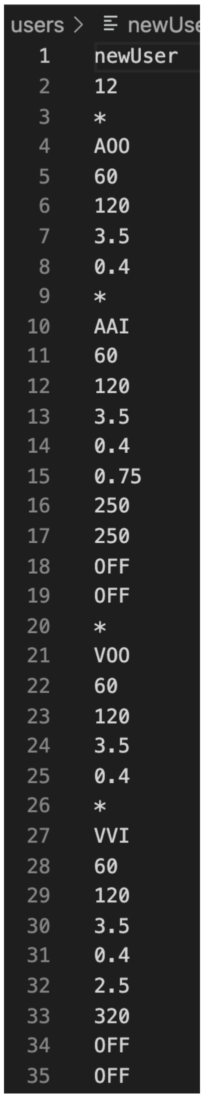
sendAOOData()
This function will take the array of parameters that is initialized through the data that is received from the pacemaker through serial communication. After words, it will take all the new saved values of the parameters from the DCM and update this array with the new values. Then the method writeListOfValues(params) will be called to transfer this data from the DCM to the pacemaker.

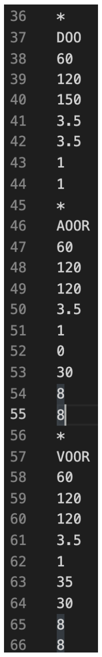
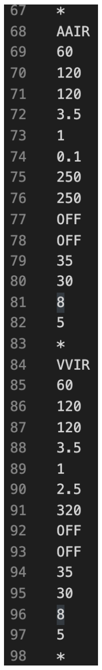
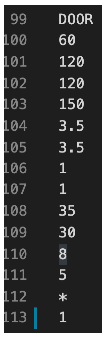
saveChanges()
This is the most important function in the class as we first define 3 different ranges for which the 4 programmable parameters can fall under. If all the parameters fall under the same ranges and increments that they should be at then the values are saved and the function updateFile() is called to permanently save those changes. If the parameters entered do not align with the range and increment that it was set for, an error message will pop up when the user saves the new parameters.

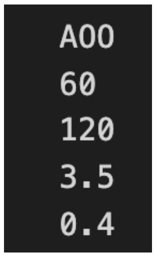
Private functions and global variables were not used in this class

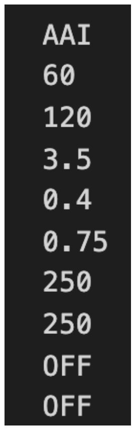
VOO, AAI, VVI, and DOO, AOOR, VOOR, AAIR, VVIR, and DOOR have their own classes representing each page of programmable parameters. All these classes are identical to the AOO page class, the only difference being that different programmable parameters are being changed in each class with different ranges and increments. The text boxes that can be edited for each class will be different as the programmable parameters are different. The logic of how data is stored and saved is the exact same throughout. Finally all functions used in the AOO class are the same in the other three classes as well with the minor changes.

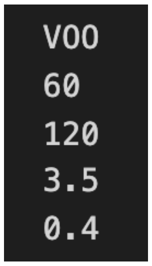
Local Storage

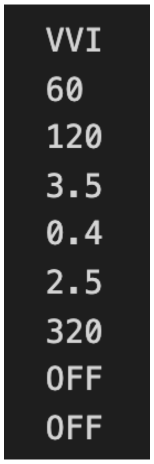
How users are Stored	 users Adarsh.txt CurrUser.txt DefaultUser.txt masterlist.txt	To store users and persist their parameter changes for all pacemaker modes, there is a 'users' subdirectory in the root of our dcm directory. This folder contains all necessary text files to create, maintain, and store users locally.
Master List	 users > masterlist.txt 1 Adarsh,newUser	The masterlist.txt file is used to store all existing usernames. This list is cross referenced when users try to log in to ensure users can only log in if they have a valid account on the system.
Sample User File	 users > newUse 1 newUser 2 12 3 * 4 AOO 5 60 6 120 7 3.5 8 0.4 9 * 10 AAI 11 60 12 120 13 3.5 14 0.4 15 0.75 16 250 17 250 18 OFF 19 OFF 20 * 21 VOO 22 60 23 120 24 3.5 25 0.4 26 * 27 VVI 28 60 29 120 30 3.5 31 0.4 32 2.5 33 320 34 OFF 35 OFF	This sample user file contains all necessary user data. Each major section is separated by an asterisk for parsability when trying to retrieve data. Each user's file name is set as their username+'.txt' so the file can be accessed directly with their input. Section 1 (lines 1 and 2) This section contains the username and password so the user can be authorized on sign in. When a user tries to log in, the masterlist is searched for the user. If the user is found, their username is used to get their user file, and the first two lines are read in order to validate their password. Section 2 (lines 3-9) → AOO parameters Section 3 (lines 9-20) → AAI parameters Section 4 (lines 20-26) → VOO parameters Section 5 (lines 26-36) → VVI parameters Section 6 (lines 36-45) → DOO parameters Section 7 (lines 45-56) → AOOR parameters Section 8 (lines 56-67) → VOOR parameters Section 9 (lines 67-83) → AAIR parameters Section 10 (lines 83-98) → VVIR parameters Section 11 (lines 98-112) → DOOR parameters These 10 sections are used to store all mode data. *Please refer to the detailed breakdown of each mode below for more information on how each mode is structured. Section 12 This single value section is the pacemaker ID

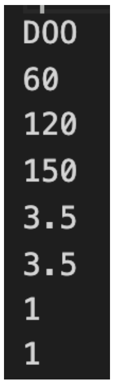
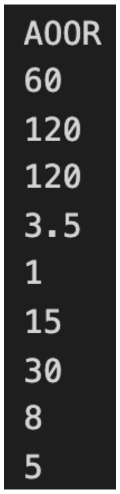
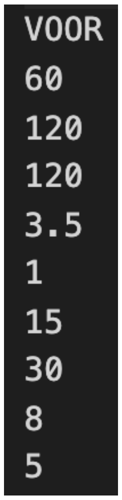
	 `*` DOO 60 120 150 3.5 3.5 1 1 `*` AOOR 60 120 120 3.5 1 0 30 8 8 `*` VOOR 60 120 120 3.5 1 35 30 8 8  `*` AAIR 60 120 120 3.5 1 0.1 250 250 OFF OFF 35 30 8 5 `*` VVIR 60 120 120 3.5 1 2.5 320 OFF OFF 35 30 8 5 `*`  DOOR 60 120 120 150 3.5 3.5 1 1 35 30 8 5 `*` 1	associated with the user. Each pacemaker is given a unique ID and when the user checks pacemaker connection, the DCM checks if the current pacemaker connected is the same as the one saved in the user's profile.

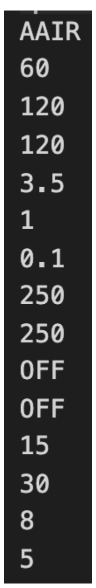
AOO	
	```
	AOO
	60
	120
	3.5
	0.4
	```
		Each line stores a different parameter
		
		Line 1: The mode (AOO)
		Line 2: Lower Rate Limit
		Line 3: Upper Rate Limit
		Line 4: Atrial Amplitude
		Line 5: Atrial Pulse Width

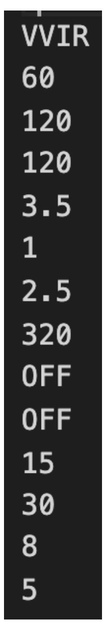
AAI	
	```
	AAI
	60
	120
	3.5
	0.4
	0.75
	250
	250
	OFF
	OFF
	```
		Each line stores a different parameter
		
		Line 1: The mode (AAI)
		Line 2: Lower Rate Limit
		Line 3: Upper Rate Limit
		Line 4: Atrial Amplitude
		Line 5: Atrial Pulse Width
		Line 6: Atrial Sensitivity
		Line 7: ARP
		Line 8: PVARP
		Line 9: Hysteresis
		Line 10: Rate Smoothing

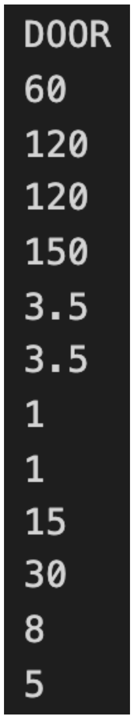
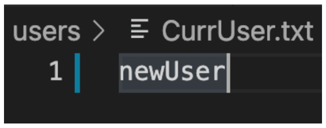
VOO	
	```
	VOO
	60
	120
	3.5
	0.4
	```
		Each line stores a different parameter
		Line 1: The mode (VOO)
		Line 2: Lower Rate Limit
		Line 3: Upper Rate Limit
		Line 4: Ventricular Amplitude
		Line 5: Ventricular Pulse Width

VVI	
	```
	VVI
	60
	120
	3.5
	0.4
	2.5
	320
	OFF
	OFF
	```
		Each line stores a different parameter
		Line 1: The mode (VVI)
		Line 2: Lower Rate Limit
		Line 3: Upper Rate Limit
		Line 4: Ventricular Amplitude
		Line 5: Ventricular Pulse Width
		Line 6: Ventricular Sensitivity
		Line 7: VRP
		Line 8: Hysteresis
		Line 9: Rate Smoothing

DOO	 DOO 60 120 150 3.5 3.5 1 1	Each line stores a different parameter Line 1: The mode (DOO) Line 2: Lower Rate Limit Line 3: Upper Rate Limit Line 4: Fixed AV delay Line 5: Atrial Amplitude Line 6: Ventricular Amplitude Line 7: Atrial Pulse Width Line 8: Ventricular Pulse width
AOOR	 AOOR 60 120 120 3.5 1 15 30 8 5	Each line stores a different parameter Line 1: The mode (AOOR) Line 2: Lower Rate Limit Line 3: Upper Rate Limit Line 4: Max sensor Rate Line 5: Atrial Amplitude Line 6: Atrial pulsewidth Line 7: Activity Threshold Line 8: Reaction time Line 9: Response factor Line 10: Recovery time
VOOR	 VOOR 60 120 120 3.5 1 15 30 8 5	Each line stores a different parameter Line 1: The mode (VOOR) Line 2: Lower Rate Limit Line 3: Upper Rate Limit Line 4: Max sensor Rate Line 5: Ventricular Amplitude Line 6: Ventricular pulse width Line 7: Activity Threshold Line 8: Reaction time Line 9: Response factor Line 10: Recovery time

AAIR	
	```
	AAIR
	60
	120
	120
	3.5
	1
	0.1
	250
	250
	OFF
	OFF
	15
	30
	8
	5
	```
		Each line stores a different parameter
		
		Line 1: The mode (AAIR)
		Line 2: Lower Rate Limit
		Line 3: Upper Rate Limit
		Line 4: Max sensor Rate
		Line 5: Atrial Amplitude
		Line 6: Atrial pulsewidth
		Line 7: Atrial sensitivity
		Line 8: ARP
		Line 9: PVARP
		Line 10: Hysteresis
		Line 11: Rate smoothing
		Line 12: Activity Threshold
		Line 13: Reaction time
		Line 14: Response factor
		Line 15: Recovery time

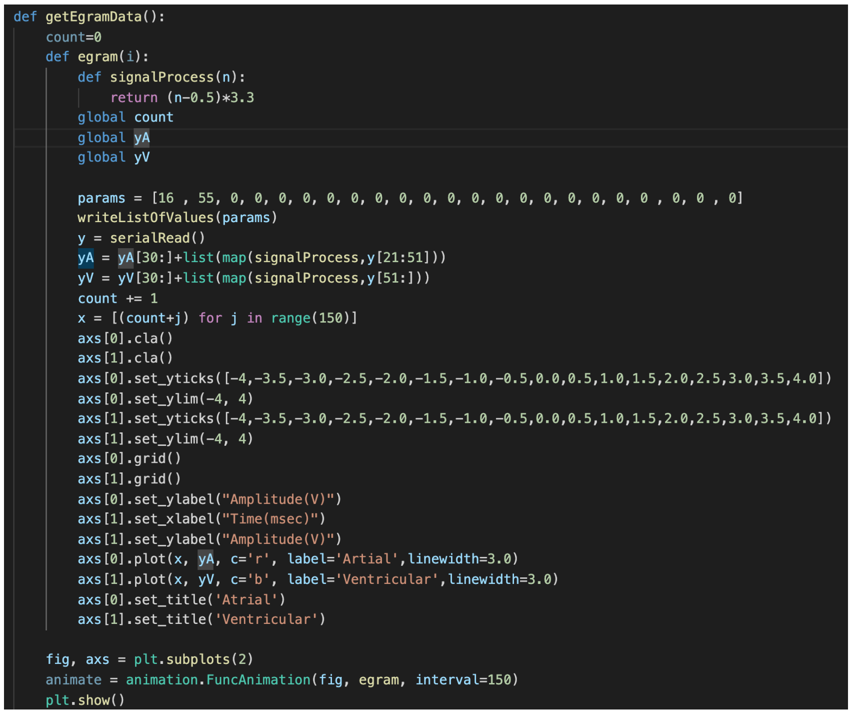
VVIR	
	```
	VVIR
	60
	120
	120
	3.5
	1
	2.5
	320
	OFF
	OFF
	15
	30
	8
	5
	```
		Each line stores a different parameter
		
		Line 1: The mode (VVIR)
		Line 2: Lower Rate Limit
		Line 3: Upper Rate Limit
		Line 4: Max sensor Rate
		Line 5: Ventricular Amplitude
		Line 6: Ventricular pulse width
		Line 7: Ventricular sensitivity
		Line 8: VRP
		Line 9: Hysteresis
		Line 10: Rate smoothing
		Line 11: Activity Threshold
		Line 12: Reaction time
		Line 13: Response factor
		Line 14: Recovery time

DOOR	 ```DOOR 60 120 120 150 3.5 3.5 1 1 15 30 8 5```	Each line stores a different parameter Line 1: The mode (DOOR) Line 2: Lower Rate Limit Line 3: Upper Rate Limit Line 4: Max sensor rate Line 5: Fixed AV delay Line 6: Atrial Amplitude Line 7: Ventricular amplitude Line 8: Atrial pulse width Line 9: Ventricular pulse width Line 10: Activity Threshold Line 11: Reaction time Line 12: Response factor Line 13: Recovery time
Current Usr File	 users > CurrUser.txt ```newUser```	This file is used to store the username of the user currently logged in. It is referenced whenever user data needs to be pulled.

Electrocardiogram Display

The egram display is a visual representation of what is happening in the heart in real time. It uses the A0 and A1 pins on the pacemaker shield to obtain the atrial and ventricular signal, respectively and displays them in a dynamic line graph. Samples are taken every 5 mssec and are plotted as they are received by the DCM.

Implementation:



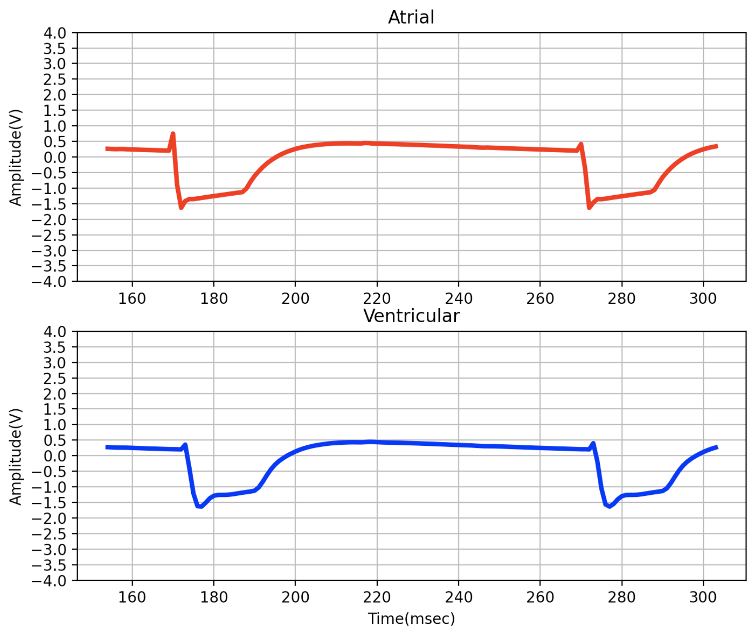
```python
def getEgramData():
    count=0
    def egram(i):
        def signalProcess(n):
            return (n-0.5)*3.3
        global count
        global yA
        global yV

        params = [16 , 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0, 0 , 0]
        writeListOfValues(params)
        y = serialRead()
        yA = yA[30:]+list(map(signalProcess,y[21:51]))
        yV = yV[30:]+list(map(signalProcess,y[51:]))
        count += 1
        x = [(count+j) for j in range(150)]
        axs[0].cla()
        axs[1].cla()
        axs[0].set_yticks([-4,-3.5,-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0])
        axs[0].set_ylim(-4, 4)
        axs[1].set_yticks([-4,-3.5,-3.0,-2.5,-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0])
        axs[1].set_ylim(-4, 4)
        axs[0].grid()
        axs[1].grid()
        axs[0].set_ylabel("Amplitude(V)")
        axs[1].set_xlabel("Time(msec)")
        axs[1].set_ylabel("Amplitude(V)")
        axs[0].plot(x, yA, c='r', label='Artial',linewidth=3.0)
        axs[1].plot(x, yV, c='b', label='Ventricular',linewidth=3.0)
        axs[0].set_title('Atrial')
        axs[1].set_title('Ventricular')

    fig, axs = plt.subplots(2)
    animate = animation.FuncAnimation(fig, egram, interval=150)
    plt.show()
```

Python's matplotlib library is used to plot the eGram data. Specifically, functional animation (FuncAnimation()) is used to animate the egram display. When a new sample is received by the DCM, it is appended to the list storing all y coordinates and older previous values are popped off. Since the pacemaker sends 30 samples that are taken 5msec apart, the plot is refreshed every 150 msec so that the graph is always displaying the data in the correct order with no overlap.

The egram() function updates the frame every 150 msec (as set in the FuncAnimation). It first requests an echo from the pacemaker, then obtains the relevant sample data. There are two lists, yA and yV, which store the data points for the atrium and ventricle, respectively. Each time new samples are obtained, those lists pop off previous values from the front and add the new values to the end, thus displaying the signal sliding from right to left. With each new frame, the previous plots are cleared and the new yA and yV lists are used to plot an updated egram. The x-axis represents time, so a counter is used to increment the x values with each frame refresh.

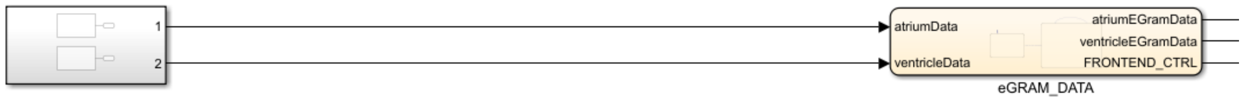
Both the atrium and ventricle activity is displayed in the same window using matplotlib's subplot functionality. This is configured in the third last line in the code block above. By specifying two subplots, two sets of graphs are created and displayed on the screen. Each of these plots (axs[0] and axs[1]) are configured in the egram function so the axes can be updated for each frame.

Sample Egram plot :



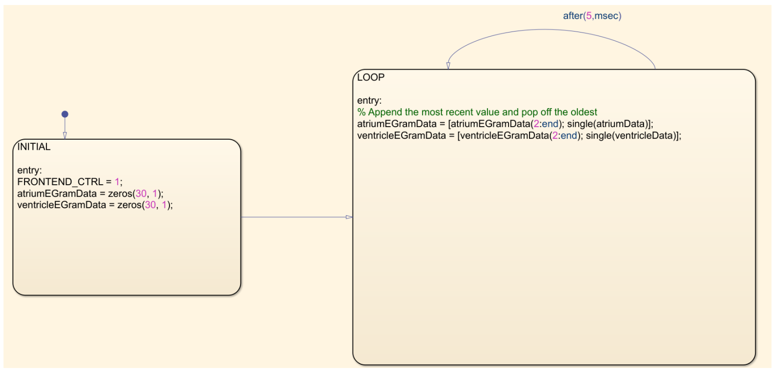
eGram Data Collection in Simulink

Collection of data samples in Simulink for the eGram data is processed within the eGRAM_DATA block of the Simulink stateflow.



On the right are the signals being sent from the pins A0 and A1, as the signals received from those pins are the electrocardio signals of the heart.

Inside the eGRAM_DATA stateflow, there are only 2 states:



```
INITIAL

entry:
FRONTEND_CTRL = 1;
atriumEGramData = zeros(30, 1);
ventricleEGramData = zeros(30, 1);
```

```
LOOP

entry:
% Append the most recent value and pop off the oldest
atriumEGramData = [atriumEGramData(2:end); single(atriumData)];
ventricleEGramData = [ventricleEGramData(2:end); single(ventricleData)];
```

The first INITIAL state is executed on simulation start, and initializes the variables atriumEGramData and ventricleEGramData to a 30 length array of zeros.

The stateflow then moves into the LOOP state, where the previously initialized array acts as a queue, and pops off the first element of the array, and pushes the data obtained from pins A0 and A1 into the end of the array. The stateflow performs this operation once every 5 milliseconds. This way, the array holds the most recent 30 samples, from the last 150 ms at any given time, and can be sent to the DCM for the electrocardiogram display.

It is also important to note that in the INITIAL state, the FRONTEND_CTRL is set to high, so that data can be retrieved from the A0 and A1 pins on the board.

DCM Testing

When the input test passed, that means that the value of that parameter was successfully saved in the data storage system. If it failed, an error message was alerted to the user saying that they had an invalid input.

Login

Example user:
Username: Adarsh
Password: 123456

User Already Exists	Input	Result
True	Username: “Adarsh” Password: “123456”	Pass: sign in to app
False	Username: “Jason” Password: “123456”	Fail: prevent sign in
True	Username: “Adarsh” Password: “password”	Fail : prevent sign in

Register

User Already Exists	Input	Result
True	Username: Adarsh Password: 123456	Fail: prevent register
False	Username: Jason Password: 123456	Pass: Allow new user register
False	Username: “ “ Password: “ “	Fail: prevent register
False	Username: “newUser“ Password: “ “	Fail: prevent register
False	Username: ““ Password: “password“	Fail: prevent register

For registration, a maximum of 10 users will be allowed to be signed in.

Lower Rate Limit (AOO, VOO, AAI, VVI, DOO, AOOR, VOOR, AAIR, VVIR, DOOR):

Lower Rate Limit cannot be greater than Upper Rate Limit.

Input	Upper Rate Limit	Input Test Result:
29	175	Fail
30	175	Pass
31	175	Fail
35	175	Pass
50	175	Pass
51	175	Pass
90	175	Pass
91	175	Fail
125	175	Pass
175	175	Pass
176	175	Fail
“test”	175	Fail
70	50	Fail

Upper Rate Limit (AOO, VOO, AAI, VVI, DOO, AOOR, VOOR, AAIR, VVIR, DOOR):

Input	Input Test Result:
49	Fail
50	Pass
55	Pass

126	Fail
175	Pass
176	Fail
“test”	Fail

Maximum Sensor Rate (AOOR, VOOR, AAIR, VVIR, DOOR):

Maximum Sensor Rate needs to be less than Upper Rate Limit

Input	Upper Rate Limit	Input Test Result:
49	175	Fail
50	175	Pass
55	175	Pass
126	175	Fail
175	175	Pass
176	175	Fail
“test”	175	Fail
175	50	Fail

Atrial Amplitude Regulated (AOO, AAI, DOO, AOOR, AAIR, DOOR) and
Ventricular Amplitude Regulated (VOO, VVI, DOO, VOOR, VVIR, DOOR):

Input	Input Test Result:
“OFF”	Pass
0	Fail
0.1	Pass
3.2	Pass
3.33	Fail
3.5	Pass
5.0	Pass
7.1	Fail
7.5	Fail
“test”	Fail

Fixed AV Delay (DOO, DOOR):

Input	Input Test Result:
“OFF”	Fail
0	Fail
50	Fail
70	Pass
71	Fail
80	Pass
300	Pass
310	Fail
301	Fail
“test”	Fail

Atrial Pulse Width (AOO, AAI, DOO, AOOR, AAIR, DOOR) and
Ventricular Pulse Width (VOO, VVI, DOO, VOOR, VVIR, DOOR):

Input	Input Test Result:
“OFF”	Fail
0	Fail
0.6	Fail
1	Pass
30	Pass
11.2	Fail
15	Pass
31	Fail
“test”	Fail

Hysteresis (AAI, VVI, AAIR, VVIR):

- For this parameter, all valid inputs for the lower rate limit are valid here as well. However this mode also needed to be tested for “OFF” input

Input	Input Test Result:
“OFF”	Pass
“off”	Fail (needs to be capital)
“no”	Fail
“Any word”	Fail

Rate Smoothing (AAI, VVI, AAIR, VVIR):

Input	Input Test Result:
“OFF”	Pass
“test”	Fail
1	Fail
3	Pass
4	Fail
6	Pass
7	Fail
9	Pass
10	Fail
12	Pass
13	Fail
15	Pass
16	Fail
18	Pass
19	Fail
21	Pass

23	Fail
25	Pass
26	Fail

Atrial Sensitivity (AAI, AAIR) and Ventricular Sensitivity (VVI, VVIR):

Input	Input Test Result:
“test”	Fail
0.1	Pass
0.25	Fail
3	Pass
5	Pass
5.1	Fail

ARP (AAI, AAIR), VRP (VVI, VVIR) and PVARP (AAI, AAIR):

Input	Input Test Result:
“test”	Fail
70	Fail
149	Fail
150	Pass
160	Pass
176	Fail
500	Pass
510	Fail
564	Fail

Activity Threshold (AOOR, VOOR, AAIR, VVIR, DOOR):

This had a drop down menu in place for the selection. Therefore, this does not require any input validation as the user can only select a valid input.

Reaction Time (AOOR, VOOR, AAIR, VVIR, DOOR):

Input	Input Test Result:
“test”	Fail
0	Fail
9	Fail
10	Pass
30	Pass
31	Fail
50	Pass
51	Fail
60	Fail

Response Factor (AOOR, VOOR, AAIR, VVIR, DOOR):

Input	Input Test Result:
“test”	Fail
0	Fail
0.1	Fail
1	Pass
10	Pass
10.5	Fail
16	Pass
17	Fail

Recovery Time (AOOR, VOOR, AAIR, VVIR, DOOR):

Input	Input Test Result:
“test”	Fail
0	Fail
0.1	Fail
1	Fail
2	Pass
10.5	Fail
16	Pass
17	Fail

Serial Communication Protocol

Note: This does not include the UART start / stop bits

Serial Communication Byte Packing Order and Parameter Types

Byte Number	Programmable Parameter	Data Type in Simulink	Range	Mapping
1	N / A	uint8	0 - 255	SYNC
2	N / A	uint8	0 - 255	34 - Set Simulink parameters to new values set by the following bits 55 - Echo the current values of the parameters to the DCM
3	UUID (Unique User Identifier)	uint8	0 - 255	This will be different for every Pacemaker A UUID of '0' indicates that the Pacemaker is ownerless
4	Pacing Mode	uint8	0 - 255	0 - Unmapped 1 - AOO Mode 2 - VOO Mode 3 - AAI Mode 4 - VVI Mode 1 - AOOR Mode 3 - AAIR Mode 2 - VOOR Mode 4 - VVIR Mode 5 - DOO Mode 5 - DOOR Mode Default value - 1
5	Lower Rate Limit	uint8	0 - 255	Default value - 60
6 7 8 9	Ventricular Amplitude	single	-3.4e38 - 3.4e38	Default value - 5.0
10	Ventricular Pulse Width	uint8	0 - 255	Default value - 1
11 12 13 14	Atrial Amplitude	single	-3.4e38 - 3.4e38	Default value - 5.0
15	Atrial Pulse Width	uint8	0 - 255	Default value - 1
16	Upper Rate Limit	uint8	0 - 255	Default value - 120

17	ARP	uint16	0 - 65535	Default value - 320
18				
19	VRP	uint16	0 - 65535	Default value - 250
20				
21	Atrial Sensitivity	single	-3.4e38 - 3.4e38	Default value - 0.75
22				
23				
24				
25	Ventricular Sensitivity	single	-3.4e38 - 3.4e38	Default value - 2.5
26				
27				
28				
29	PVARP	uint16	0 - 65535	Default value - 250
30				
31	Hysteresis	uint8	0 - 255	Default value - 0 (OFF)
32	Rate Smoothing	uint8	0 - 255	Default value - 0 (OFF)
33	Maximum Sensor Rate	uint8	0 - 255	Default value - 120
34	Fixed AV Delay	uint16	0 - 65536	Default value - 150
35				
36	Activity Threshold	uint8	0 - 255	0 - Very Low 5 - Low 10 - Medium Low 15 - Medium 20 - Medium High 30 - High 35 - Very High Default value - 10
37	Reaction Time	uint8	0 - 255	Default value - 30 (sec)
38	Response Factor	uint8	0 - 255	Default value - 8
39	Recovery Time	uint8	0 - 255	Default value - 5 (min)

Description

The stream of data sent between the DCM and Pacemaker consists of the 39 bytes outlined in the table above, and the echo function that sends parameters from the pacemaker to the DCM consists of an additional 240 bytes of sample data from the A0 and A1 pins. The first byte of our stream is a sync bit that always has a value of 16 and indicates the start of the stream The second bit is the function bit and it tells the pacemaker what function to perform with the data. If the second byte is 34 the pacemaker is set to the parameters being passed in. If the second byte is 55, the pacemaker echos its current parameters back to the DCM. The third byte is a UUID which we use to identify each pacemaker. The fourth byte is the pacing mode. The rest of the stream consists of the parameters outlined above.

Setting Parameters

DCM

```
def writeListOfValues(values):
   pattern ="<BBBBBfBfBBHHffHBBBHBBBB"
   assert (len(pattern[1:]) == len(values)), "Pattern and values are not equal size"
   endian = pattern[0]
   pattern = pattern[1:]
   packet = []
   for i in range(0, len(values)):
       packedData = pack((endian + pattern[i]), values[i])
       packet.append(packedData)
   for pac in packet:
       ser.write(pac)

params = [16, 34, 0, 1, 60, 3.5, 1, 3.5, 1, 120, 250, 320, 0.001, 0.0025, 250, 0, 0,
120, 150, 15, 30, 8, 5]
writeListOfValues(params)
```

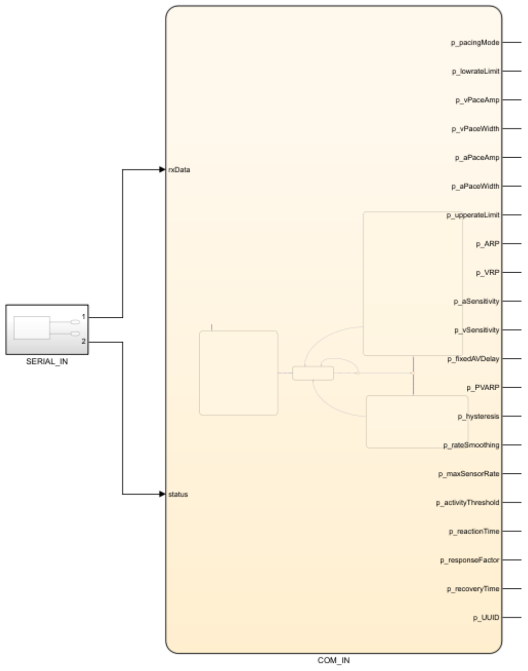
To set parameters, the writeListOfValues() takes an array of values and sends each one after packing it. In the second line, the pattern specifies types of each byte that's being sent in the stream. According to the python 'structs' library, the type mappings we used are as follows:

unit8	B
unit16	H
single	f

The '<' at the start of the stream indicates the data was packed using little endian. To specify this stream should be set on the pacemaker, the second element in the list of values is set to 34.

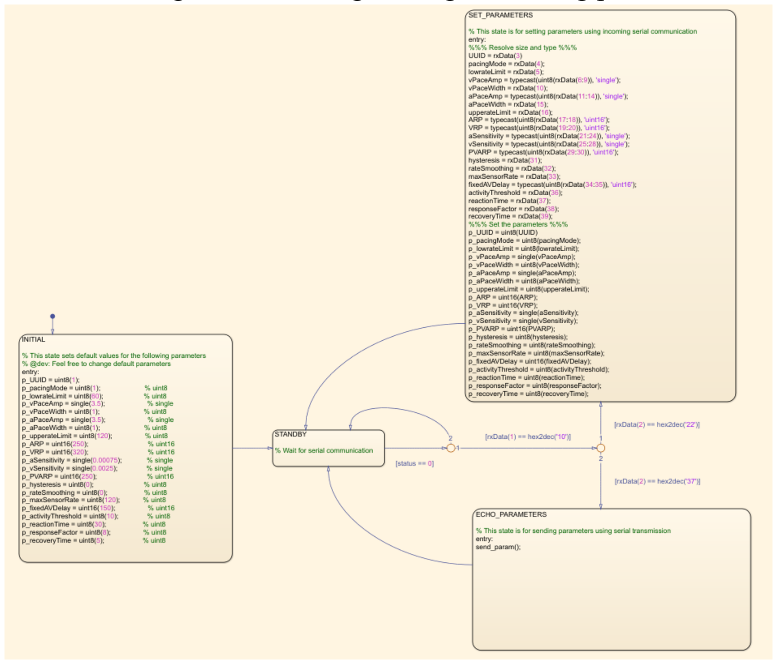
Simulink

Within Simulink, all data related to serial communication is processed within the COM_IN block (shown below):



This stateflow receives the serial data and status signals from the subsystem SERIAL_IN, which has the Serial Receive block from the FRDM K64F support package, that allows our DCM to communicate with our stateflow.

Inside the COM_IN block is the logic for receiving, setting or echoing parameters:


[status == 0]
[rxData(1) == hex2dec('10')]
[rxData(2) == hex2dec('22')]
[rxData(2) == hex2dec('37')]
1
2
1
2

```
INITIAL

% This state sets default values for the following parameters
% @dev: Feel free to change default parameters
entry:
p_UUID = uint8(1);
p_pacingMode = uint8(1);                % uint8
p_lowrateLimit = uint8(60);             % uint8
p_vPaceAmp = single(3.5);               % single
p_vPaceWidth = uint8(1);                % uint8
p_aPaceAmp = single(3.5);               % single
p_aPaceWidth = uint8(1);                % uint8
p_upperateLimit = uint8(120);           % uint8
p_ARP = uint16(250);                    % uint16
p_VRP = uint16(320);                    % uint16
p_aSensitivity = single(0.00075);       % single
p_vSensitivity = single(0.0025);        % single
p_PVARP = uint16(250);                  % uint16
p_hysteresis = uint8(0);                % uint8
p_rateSmoothing = uint8(0);             % uint8
p_maxSensorRate = uint8(120);           % uint8
p_fixedAVDelay = uint16(150);           % uint16
p_activityThreshold = uint8(10);        % uint8
p_reactionTime = uint8(30);             % uint8
p_responseFactor = uint8(8);            % uint8
p_recoveryTime = uint8(5);              % uint8
```

```
STANDBY

% Wait for serial communication
```

```
SET_PARAMETERS

% This state is for setting parameters using incoming serial communication
entry:
%%% Resolve size and type %%%
UUID = rxData(3)
pacingMode = rxData(4);
lowrateLimit = rxData(5);
vPaceAmp = typecast(uint8(rxData(6:9)), 'single');
vPaceWidth = rxData(10);
aPaceAmp = typecast(uint8(rxData(11:14)), 'single');
aPaceWidth = rxData(15);
upperateLimit = rxData(16);
ARP = typecast(uint8(rxData(17:18)), 'uint16');
VRP = typecast(uint8(rxData(19:20)), 'uint16');
aSensitivity = typecast(uint8(rxData(21:24)), 'single');
vSensitivity = typecast(uint8(rxData(25:28)), 'single');
PVARP = typecast(uint8(rxData(29:30)), 'uint16');
hysteresis = rxData(31);
rateSmoothing = rxData(32);
maxSensorRate = rxData(33);
fixedAVDelay = typecast(uint8(rxData(34:35)), 'uint16');
activityThreshold = rxData(36);
reactionTime = rxData(37);
responseFactor = rxData(38);
recoveryTime = rxData(39);
%%% Set the parameters %%%
p_UUID = uint8(UUID)
p_pacingMode = uint8(pacingMode);
p_lowrateLimit = uint8(lowrateLimit);
p_vPaceAmp = single(vPaceAmp);
p_vPaceWidth = uint8(vPaceWidth);
p_aPaceAmp = single(aPaceAmp);
p_aPaceWidth = uint8(aPaceWidth);
p_upperateLimit = uint8(upperateLimit);
p_ARP = uint16(ARP);
p_VRP = uint16(VRP);
p_aSensitivity = single(aSensitivity);
p_vSensitivity = single(vSensitivity);
p_PVARP = uint16(PVARP);
p_hysteresis = uint8(hysteresis);
p_rateSmoothing = uint8(rateSmoothing);
p_maxSensorRate = uint8(maxSensorRate);
p_fixedAVDelay = uint16(fixedAVDelay);
p_activityThreshold = uint8(activityThreshold);
p_reactionTime = uint8(reactionTime);
p_responseFactor = uint8(responseFactor);
p_recoveryTime = uint8(recoveryTime);
```

```
ECHO_PARAMETERS

% This state is for sending parameters using serial transmission
entry:
send_param();
```

The stateflow begins in the INITIAL block, which is executed on simulation start. This block is responsible for setting the default values of our programmable parameters. The default values of each programmable parameter are listed in the **Serial Communication Byte Packing Order and Parameter Types** table above. From the INITIAL state, the stateflow then moves to the STANDBY state, where it waits for data sent by serial communication. The Serial Receive block from the FRDM K64F support package outputs 2 signals, the rxData, which is the 39 bytes of data we expect from the DCM, and the status signal, which is 32 when there is no inbound communication, and 0 if there are the 39 received bytes from serial communication. When status goes to 0, the stateflow goes to a breakpoint which loops back to the STANDBY state if the first byte in the serial communication is not 16 (or 0x10 in hex). This first byte is the "Sync byte" and protects our stateflow from executing unwanted actions if, for example, the serial communication is not the correct format.

After the sync byte, the next byte from the serial communication determines if the stateflow should set or echo parameters, and this is controlled by the second byte being either a 34 (or 0x22) for setting parameters, or 55 (or 0x37) for echoing them. In the case of the second byte being 34 then the stateflow will take bytes 3 through 39 and assign them to the correct parameters. Some parameters, such as aPaceAmp or ARP are of type single or uint16, so they will take up 2 or 4 bytes as compared to other parameters of type uint8. Once the new parameters are set, since the parameters are listed as output data of this stateflow, the new values can now be used for the pacing modes. If the second byte is 55 instead, then the stateflow will call the global Simulink function "send_param()" which will echo the current values of all the programmable parameters.

After these two states, the stateflow loops back to the standby state to await the next serial communication attempt.

Echo Parameters

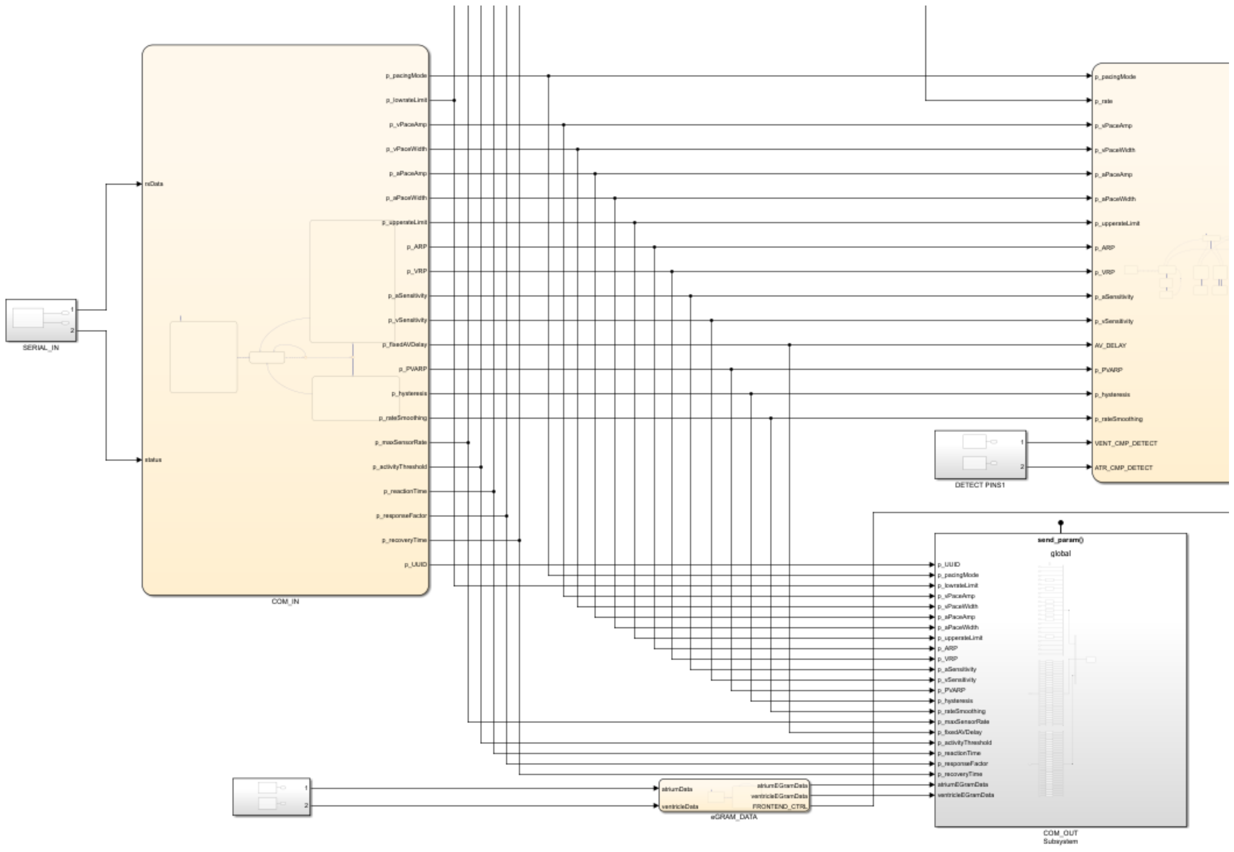
DCM

```
def serialRead ():
   pattern =
"<BBBfBfBBhhffhBBBhBBBBffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff"
   res = ser.read(278)
   res = list(unpack(pattern, res))
   return res

params = [16 , 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 , 0, 0 , 0]
writeListOfValues(params)
echoRes = serialRead()
```

Before calling the serialRead() function to retrieve the echoed parameters, a stream of data is sent with the function set to 55 and each parameter set to 0. This empty stream of data is used as a request for an echo from the pacemaker. To unpack the stream received by the DCM, the serialRead() function uses a similar pattern of variable types as writeListOfValues() , but does not retrieve the sync and function bytes and retrieves an additional 240 bytes of sample data. The sample data is all sent as type single and there are a total of 60 samples sent. The first 30 represent the output of the atrium and the next 30 represent the ventricle.

Simulink

For echoing parameters in Simulink, after the “send_param()” global function is called by having the second byte of rxData being 55 (as mentioned above), the Simulink function will take the current parameters and eGram data and transmit it back to the DCM.



The “send_param()” function is the block labeled COM_OUT which accepts input from signals from the COM_IN stateflow, and the eGRAM_DATA block. Inside the function, the data is processed and packed as necessary before being transmitted using the Serial Transmit block from the FRDM K64F support package.

As shown in the image to the right, the input is sent into the function from the output of COM_IN and eGRAM_DATA blocks. Data that comes from the COM_IN block that is more than one byte, ie, uint16 or single data types are packed into uint8 vectors (using the Data Pack Simulink block), which are then passed into a MUX with 21 inputs, which collects the data to be sent. Data coming in from the eGRAM_DATA block comes into the "send_param()" function in the form of a 30 row single array, so it must be passed through a DEMUX and then each sample must be packed into uint8 vectors individually before being put into a 30 input MUX to be transmitted. Since there are 37 bytes of programmable parameters, plus 30 single samples of atrial data, and 30 single samples of ventricular data, this means that 277 bytes of data will be transmitted back to the DCM.