

SFWRENG 3K04

Part 1 Documentation - Pacemaker Design

Group #12

Matt Mione, Riley Mione, Derek Paylor, Johnathan Spinelli, Laura Yang

Table of Contents

Simulink Model Overview	3
Section 1: Serial Communication	5
1.1. Serial Receive	5
1.1.2. Serial Receive Block	6
1.1.2. Stateflow	6
1.1.3. Outputs	8
1.2. Serial Transmit	9
1.2.1 Multiplexer	10
1.3 Design Decisions	10
1.4 Likely Changes to Requirements and Design	11
Section 2: Pacing Stateflow	11
2.1. Initial State	12
2.2. AOO Stateflow	13
2.3. VOO Stateflow	14
2.4. AAI Stateflow	15
2.5. VVI Stateflow	16
2.6. DOO Stateflow	17
2.7. AOOR Stateflow	18
2.8. VOOR Stateflow	18
2.9. AAIR Stateflow	19
2.10. VVIR Stateflow	20
2.11. DOOR Stateflow	20
2.12. Design Decisions	21
2.13. Likely Changes to Requirements and Design	21

Section 3: K64F Hardware Hiding	21
3.1. Inputs	22
3.2. Outputs	23
3.3. Design Decisions	24
3.4. Likely Changes to Requirements and Design	24
Section 4: Rate Adaptivity	24
4.1. Accelerometer Signal Processing	25
4.2. Varying the Rate Based on Accelerometer Data	28
4.2.1. Computing Threshold Values	29
4.2.2. Approximating Linear Functions with Discrete Steps	31
4.2.3. Rate Adaptivity Stateflow	34
4.3. Design Decisions	36
4.4. Likely Changes to Requirements and Design	36
Section 5: Auxiliary Blocks	37
5.1. Amplitude to Duty Cycle (3.3V)	37
5.2. Amplitude to Duty Cycle (5V)	37
5.3. BPM to Period	38
5.4. Design Decisions	38
5.5. Likely Changes to Requirements and Design	38
Appendix A: Blank Simulink Model	39

Simulink Model Overview

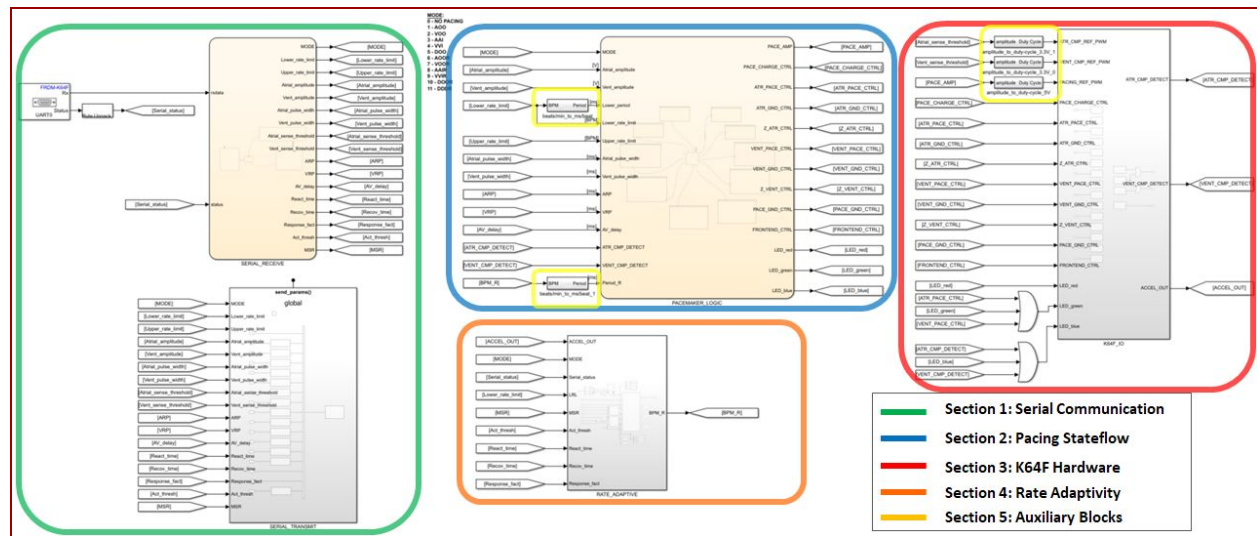


Figure 1: Model Overview

As an introduction to understanding the Simulink model, it can be roughly broken up into sections, shown in Figure 1 above. Each section will have a corresponding entry in this document (see Table of Contents) with greater detail on its function and design.

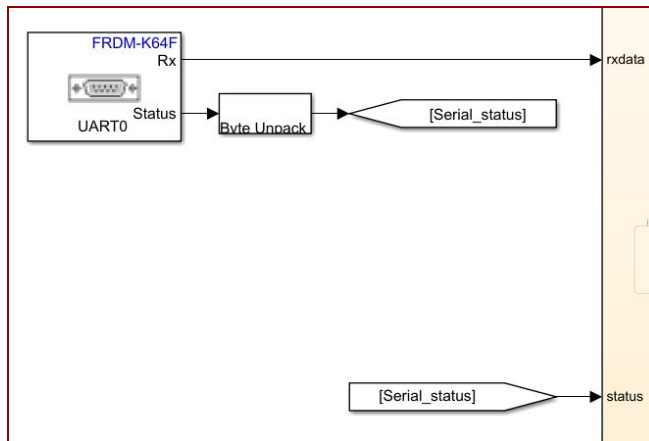
Section 1 consists of the implementation of serial communication with the DCM, which is responsible for reading programmable parameters from the DCM and then sending those parameters back as a confirmation that they were received correctly. The primary Stateflow chart and associated state logic will comprise Section 2, which use inputs to determine the necessary timing and pin controls in hardware to form the function of the pacemaker. Section 3 contains the input and output hardware components for the K64F microcontroller, which are read from and written to by Stateflow and serial input parameters. All modules relating to implementation of pacing rate adaptivity are described in Section 4. Finally, Section 5 consists of any remaining blocks in the model space, which perform simple calculations or unit conversions for various components in the model to use.

One overarching design decision for the model was to make extensive use of 'Goto' and 'From' blocks to transfer signals between modules rather than connecting them directly, for organizational purposes.

For an unobstructed view of the entire model, see Appendix A.

1.1.2. Serial Receive Block

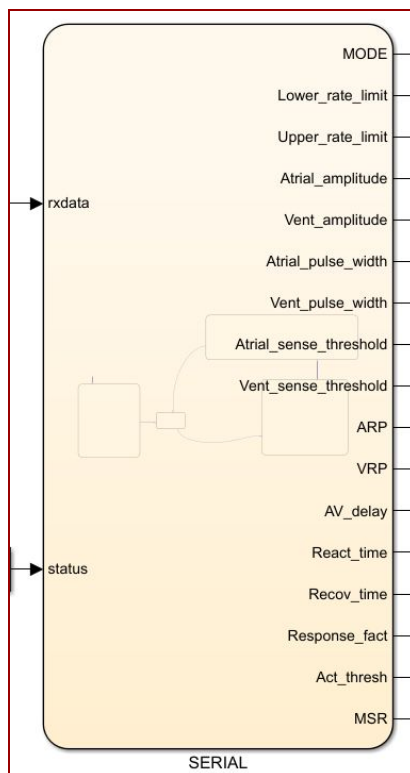
Figure 4: FRDM-K64F Block



The FRDM-K64F serial receive blocks gives two outputs, rxdata and status. The 'Byte Unpack' block converts status from a 1x1 matrix to an integer for the stateflow to use.

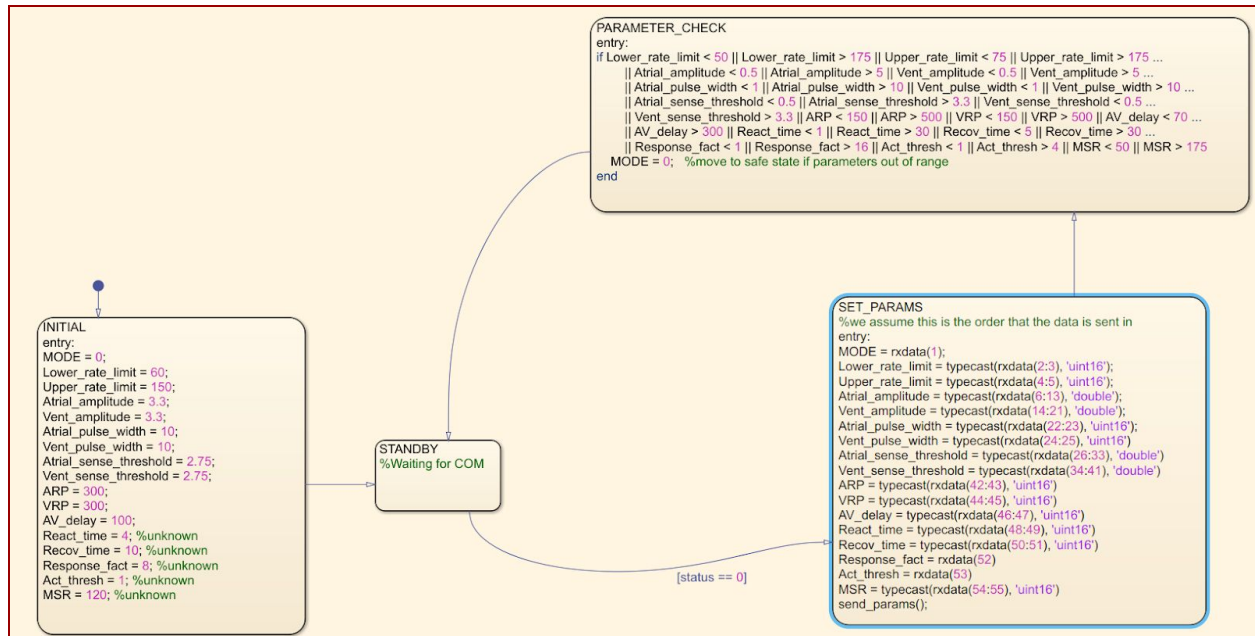
1.1.2. Stateflow

Figure 2: Serial Receive Chart



The stateflow takes two inputs, 'rxdata' and 'status', and gives outputs that are used in other parts of the model. 'rxdata' is received as a vector and values will be indexed from the vector and assigned to its corresponding variable within the chart.

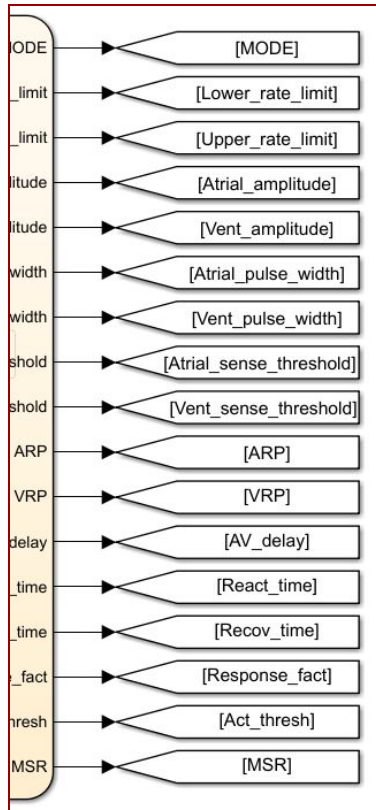
Figure 5: Serial Communication Stateflow



Beginning in the 'INITIAL' (default) state, parameters are set to nominal values and the pacemaker is initiated in standby mode (ie. 'MODE' is 0). Upon successful serial read operation, the parameters are set to the values received in 'rxddata' from the DCM. No values will be set unless a successful serial read has been completed, meaning the number of bytes sent by the DCM is the same as the byte length chosen in the serial receive block. At the end of the 'SET_PARAMS' state, the function 'send_params()' is called and the parameters are sent back to the DCM to ensure the values are read correctly. In the 'PARAMETER_CHECK' state, the variables read from the DCM are compared to the bounded limits. If any value read in from the DCM is not considered safe (out of acceptable range), the pacemaker will enter a standby state from which no pacing occurs ('MODE' is 0).

1.1.3. Outputs

Figure 6: Serial Receive Outputs



The outputs to the ‘SERIAL_RECEIVE’ chart are Goto tags, which allows for a neater Simulink model. These parameters are used in the pacing stateflow, K64F hardware, and rate adaptivity charts and subsystems to define pace timing and functionality.

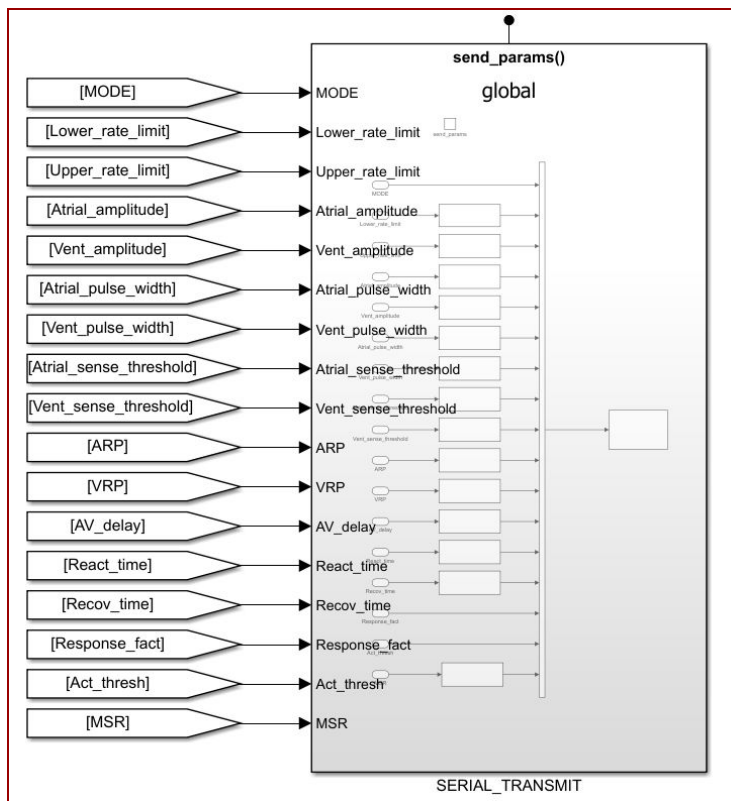
Table 1: Input Parameter Reference

Parameter Name	Units	Data type	Range / Nominal
MODE	N/A	uint8	0 - 11
Lower rate limit	BPM	uint16	50 - URL / 60BPM
Upper rate limit	BPM	uint16	75 - 175BPM / 120BPM
Atrial amplitude	V	double	0.5 - 5V / 3V
Vent amplitude	V	double	0.5 - 5V / 3V
Atrial pulse width	ms	uint16	1 - 10ms / 10ms
Vent pulse width	ms	uint16	1 - 10ms / 10ms
Atrial sense threshold	V	double	0.5 - 3.3V / 2.75V
Vent sense threshold	V	double	0.5 - 3.3V / 2.75V
Atrial refractory period	ms	uint16	150 - 500ms / 300ms

Vent refractory period	ms	uint16	150 - 500ms / 300ms
AV Delay	ms	uint16	70-300ms / 150ms
Reaction time	s	uint16	1 - 30s / 4s
Recovery time	s	uint16	5 - 30s / 10s
Response factor	N/A	uint8	1-16 / 8 1 - Slow 16 - Fast
Activity threshold	N/A	uint8	1-4 / 2 1 - Low 4 - High
Max sensor rate (MSR)	BPM	uint16	50-175BPM / 120BPM

1.2. Serial Transmit

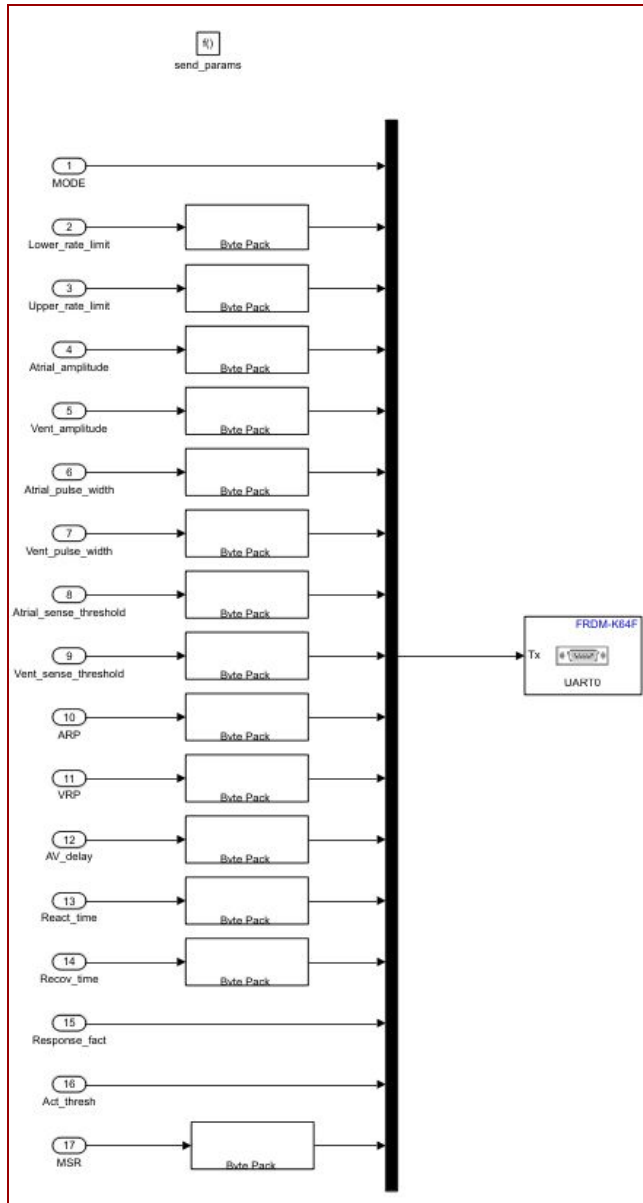
Figure 7: Serial Transmit Components



The 'send_params()' function takes inputs from all programmable parameters used by the Simulink model, in the form of 'From' tags originating from the 'SERIAL_RECEIVE' chart. The send_params function is called within the 'SERIAL_RECEIVE' stateflow and sends the parameters back to the DCM to ensure they were read into the model with the correct values that were sent.

1.2.1 Multiplexer

Figure 10: Serial Transmit Multiplexer



Within the 'send_params()' function, byte pack blocks are used to convert uint16 and doubles into uint8s which are combined into a vector and sent to the DCM using the KRDM-K64F serial transmit block.

1.3 Design Decisions

Initially in the 'SERIAL_RECEIVE' stateflow (see Figure 5), the 'SET_PARAMS' state was accompanied by a 'SEND_PARAMS' state that would call the 'send_params()' function as well as transmit electrogram data, and the two states were separated in transition by the value of an additional byte of data in the serial data string. The intention was that the DCM would be continually communicating with the pacemaker, and the additional byte could differentiate between whether the DCM is requesting to modify parameters or to receive electrogram data. Following issues with serial

transmission blocks and requirement changes for electrogram data, this plan was adjusted to the current implementation, in which the 'send_params()' function is simply called within 'SET_PARAMS' after reading the incoming data. The DCM only transmits data when its 'submit' button is pressed in the GUI, and thus will receive and read data back from the pacemaker (and verify it) right after.

A safe mode was implemented to verify that all values sent by the DCM are reasonable and allowable. This is a *safety critical system*. Thus, the assumption that all values sent to the hardware will be within a safe range is NOT a valid assumption, or a safe one. Also to increase redundancy, a state at the start of the stateflow that sets safe values preliminarily, which will be overwritten later. These design decisions came from our need to prioritize safety above all else in this project.

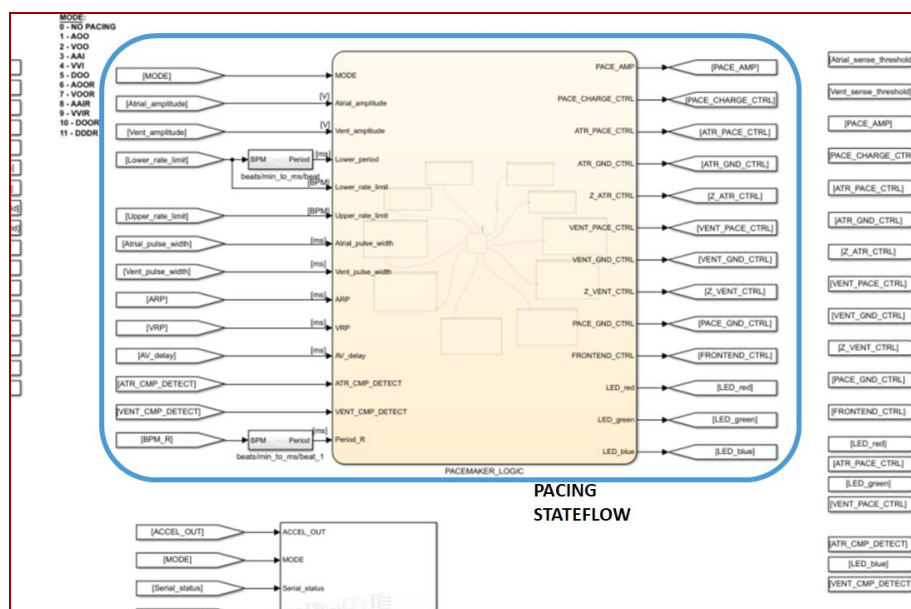
Using Simulink's built in K64F Serial Read/Transmit blocks helped simplify the design, and its functionality determined a majority of the design decisions made in the state flows and functions that rely on them. For example, the serially read variables are packed into a matrix called *rxdata*, and our stateflows manipulate the individual elements of said array to store the data and send it to the rest of our model. This would, of course, have to be changed if the Simulink's serial functions worked differently. In this sense, a majority of the design decisions were based around interfacing with those pre-built functions.

1.4 Likely Changes to Requirements and Design

If development of this project continued, sending electrogram data to the DCM would be implemented. This would require some changes to the serial communication stateflow. If the DCM was in a mode where it was to display the egram, the only communication between the DCM and the Simulink model would be sending the 'VENT_CMP_DETECT' and 'ATR_CMP_DETECT' from the model to the DCM. This would require the addition of another state that would only be reached if a certain condition sent by the DCM was met. For example, the Simulink model would know if the DCM was in the egram mode if the last byte sent was 0xFF. In this new state, parameters would only be sent to the DCM and the model would not save any of the values being received from the DCM. After exiting the egram mode, the DCM would send 0x55 as the last byte and that would signal to the Simulink model that it can now receive and save the parameters sent in from the DCM.

Section 2: Pacing Stateflow

Figure 11: Pacing Stateflow



Consisting of the 'PACEMAKER_LOGIC' chart as well as its input and output tags, the pacing stateflow determines the necessary pin outputs and timing to accomplish the pacemaker's functionality in the various pacing modes. The chart takes

input from a number of other components, including serially-received DCM parameters (see Section 1.1.3), sensing signals from the Pacemaker Shield and a variable-rate parameter from the rate adaptivity module (see Section 4).

Figure 12: Inside Pacing Chart

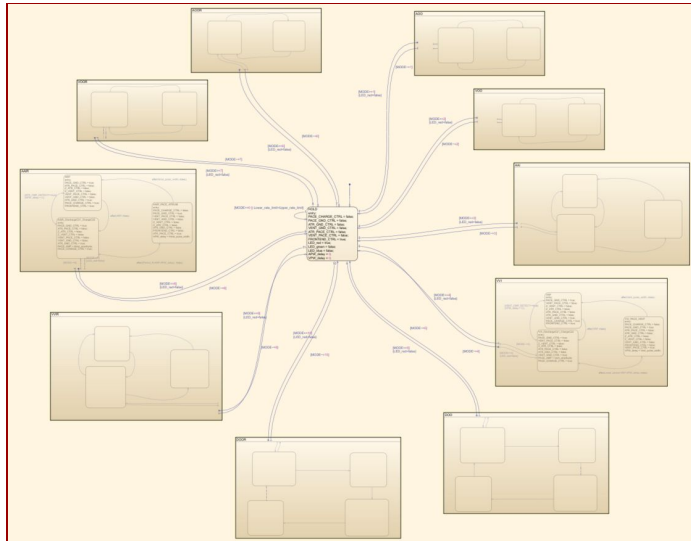


Figure 12 shows an overview of the various states (and subchart boxes) within 'PACEMAKER_LOGIC'. Individual states and modes are described in detail, including design decisions, in the following sections.

2.1. Initial State

Figure 13: Initial State

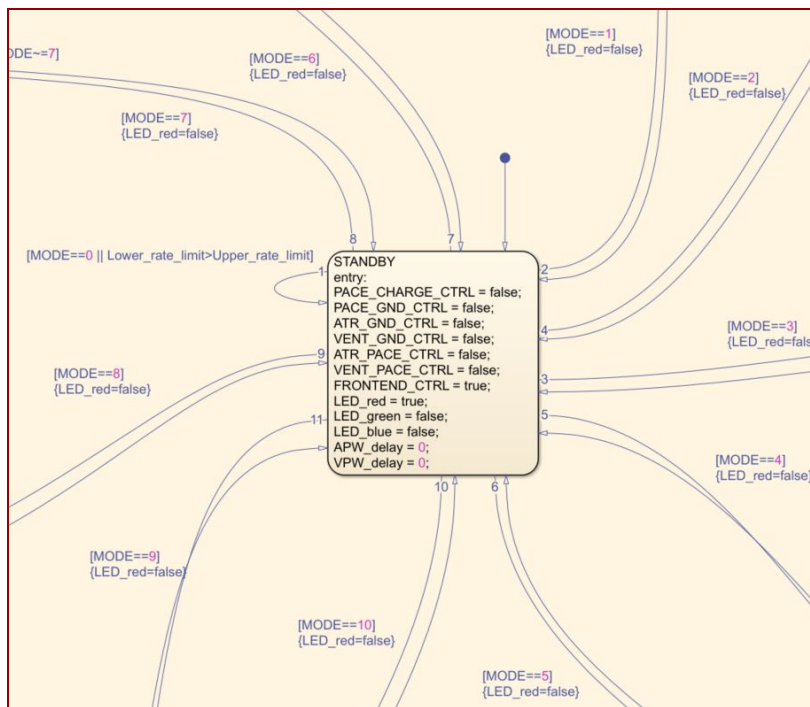


Figure 13 shows the initial (default) state that the pacing stateflow begins in. The program will remain in this state unless it receives a 'MODE' input within 1 - 10, so it mainly serves as a junction that will transition to a certain pacing mode based on the MODE input. Since no pacing occurs in this state, it is also used as a standby or safe state from which no artificial paces are made. This state opens most of the switches on the Pacemaker Shield circuitry as to provide minimal interaction with the heart, but keeps 'FRONTEND_CTRL' closed to allow for sensing data to

be sent to the DCM while in this state (this feature was not implemented for this version of the project). Since it acts as a safe state, if the program encounters a problem while reading serial data (if the incoming

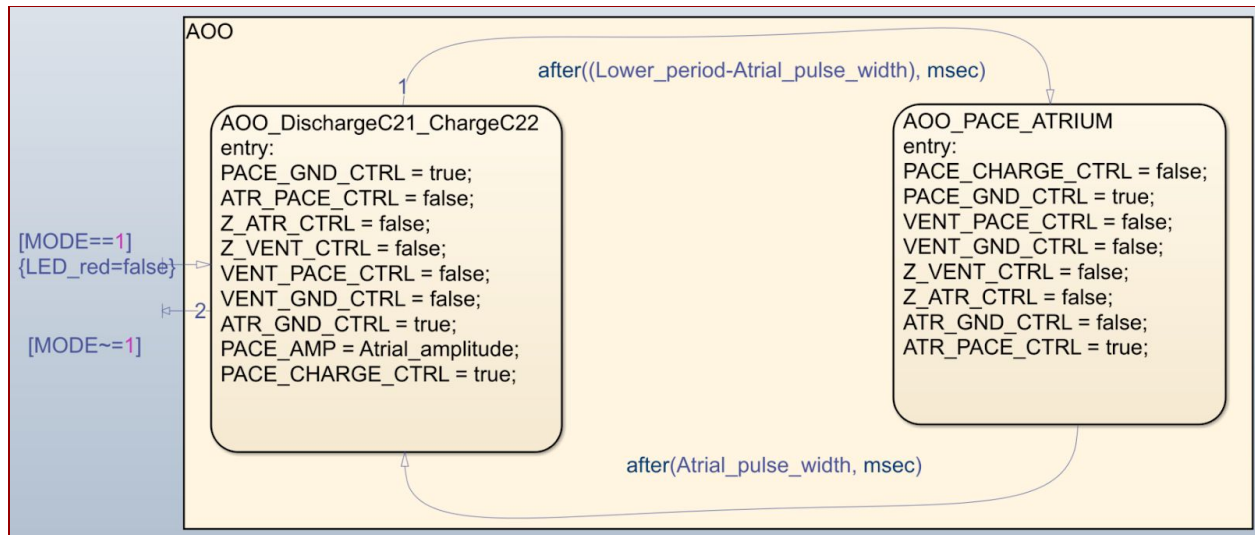
parameters are out of their acceptable range), the program will return to this state as a failsafe (see Section 1 for details on serial data checking).

As a safety feature, the program will loop indefinitely in this state if the Lower rate limit is greater than the Upper rate limit, which is checked by its first transition (upper left side of the state).

The red LED on the K64F is activated while in this state to indicate that it is in standby mode, but it is turned off during transition out of this state. Two local stateflow parameters are initialized to zero, which are used in inhibiting (XXIX) modes.

2.2. AOO Stateflow

Figure 14: AOO Stateflow



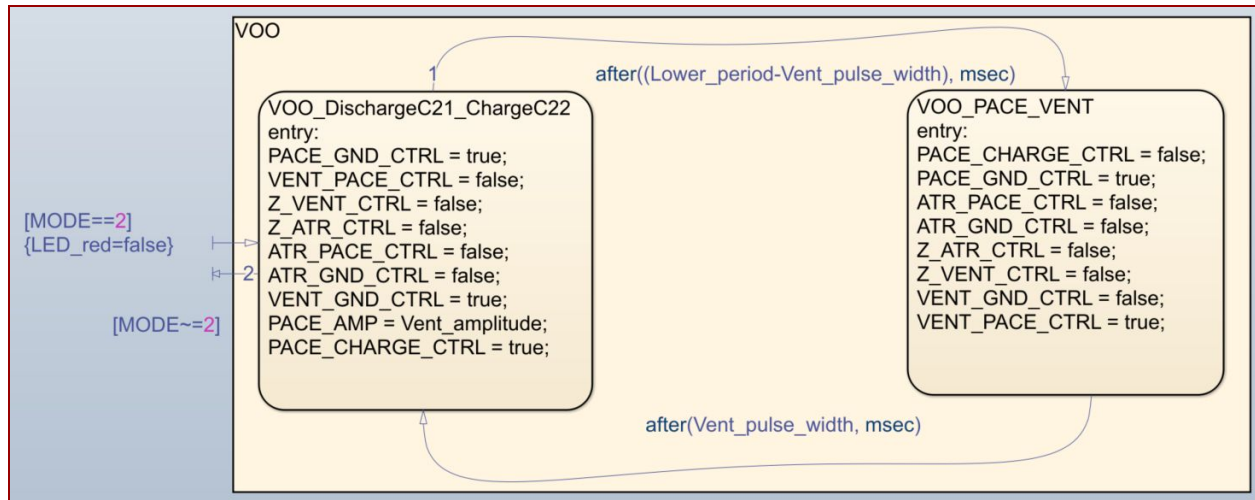
From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 1, the stateflow will enter the AOO set of states shown in Figure 14. Once entered, the AOO mode cycles between the two states shown above. Beginning in ‘AOO_DischargeC21_ChargeC22’, the output parameters are set such that the state discharges the blocking capacitor (C21) then charges the pacing capacitor (C22) to the voltage corresponding to ‘Atrial_amplitude’ (see Section 5.2 to see how ‘PACE_AMP’ adjusts the PWM signal that charges C22). While in this state, all of the LEDs are off. The program will remain in this state for the time difference between ‘Lower_period’ and ‘Atrial_pulse_width’ (ie. when the transition condition ‘after((Lower_period - Atrial_pulse_width), msec)’ is true). The ‘Lower_period’ input simply corresponds to the period of desired pacing, in milliseconds (see Section 5.3 for this conversion). For example, if the Lower rate limit is set to 60 beats per minute and ‘Atrial_pulse_width’ is set to 10ms, the period for one pace will be 1 second (‘Lower_period’ will be 1000 milliseconds), and the expression ‘Lower_period - Atrial_pulse_width’ becomes 990ms. Thus the program would remain in this state for 990ms. This state is also a safe state to transition out of if the ‘MODE’ input has been altered, hence the second transition out of this state if ‘MODE~=1’.

The other state, ‘AOO_PACE_ATRIUM’, sets output parameters as to discharge C22 into the atrium. The program remains in this state very briefly, for the duration of ‘Atrial_pulse_width’, which is typically less than 10ms. Elsewhere in the model, the green LED is set to receive ‘ATR_PACE_CTRL’ as a boolean input to indicate an artificial pace being made by the pacemaker (see Section 3).

Note that the total time to cycle through these states is equivalent to the desired period for one pace. Borrowing input values from the example described above, the pacing period would be 1s (1000ms), and the program would remain in 'AOO_DischargeC21_ChargeC22' for 990ms then 'AOO_PACE_ATRIUM' for 10ms.

2.3. VOO Stateflow

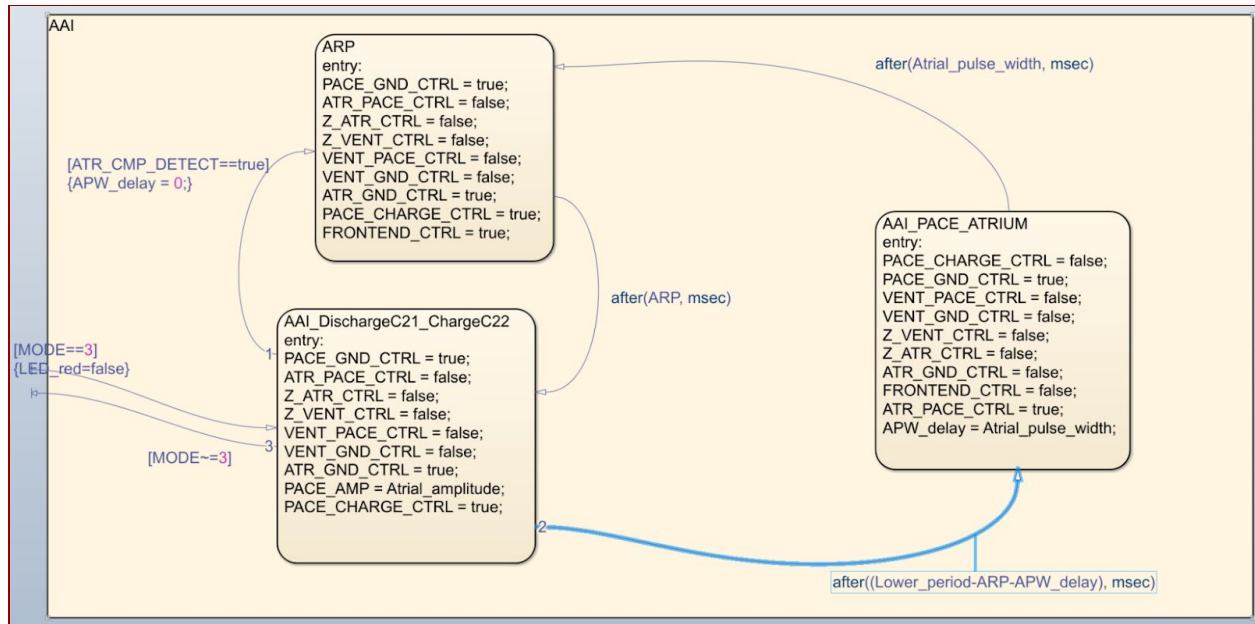
Figure 15: VOO Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 2, the stateflow will enter the VOO set of states shown in Figure 15. The functionality is very similar to AOO, albeit here it is the ventricle being paced. Thus the states and transitions follow the same logic as the AOO mode, so explanations in Section 2.2 can be used as a reference. Changes are made to pace the ventricle rather than the atrium, to transition based on 'Vent_pulse_width' rather than 'Atrial_pulse_width', and to charge C22 with the voltage of 'Vent_amplitude' rather than 'Atrial_amplitude'. Similar to atrial pacing, the green LED is set to receive 'VENT_PACE_CTRL' as a boolean input to indicate an artificial pace being made by the pacemaker.

2.4. AAI Stateflow

Figure 16: AAI Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 3, the stateflow will enter the AAI set of states shown in Figure 16. ‘AAI_DischargeC21_ChargeC22’ is similar to the corresponding state in AOO mode, in which capacitor C21 is discharging and C22 is charging, however this state is now affected by sensing natural atrial events. This can be in the additional transition if an atrial event is detected (if ‘ATR_CMP_DETECT’ is TRUE), which corresponds to detecting a natural pace, and the state will transition to ‘ARP’ (for Atrial Refractory Period) where the same parameters are set but the sensing circuit has no effect on state transitions - the program will always remain in the ‘ARP’ state for the duration of the ARP input value, regardless of sensed atrial events. The ‘ARP’ state always returns back to ‘AAI_DischargeC21_ChargeC22’, and if there is no detected atrial event within its duration, the pacemaker must make an artificial pace and transitions to ‘AAI_PACE_ATRIUM’, which sends an atrial pace in the same way as in AOO.

‘AAI_PACE_ATRIUM’ disables the sensing circuit, and also transitions into the ‘ARP’ state.

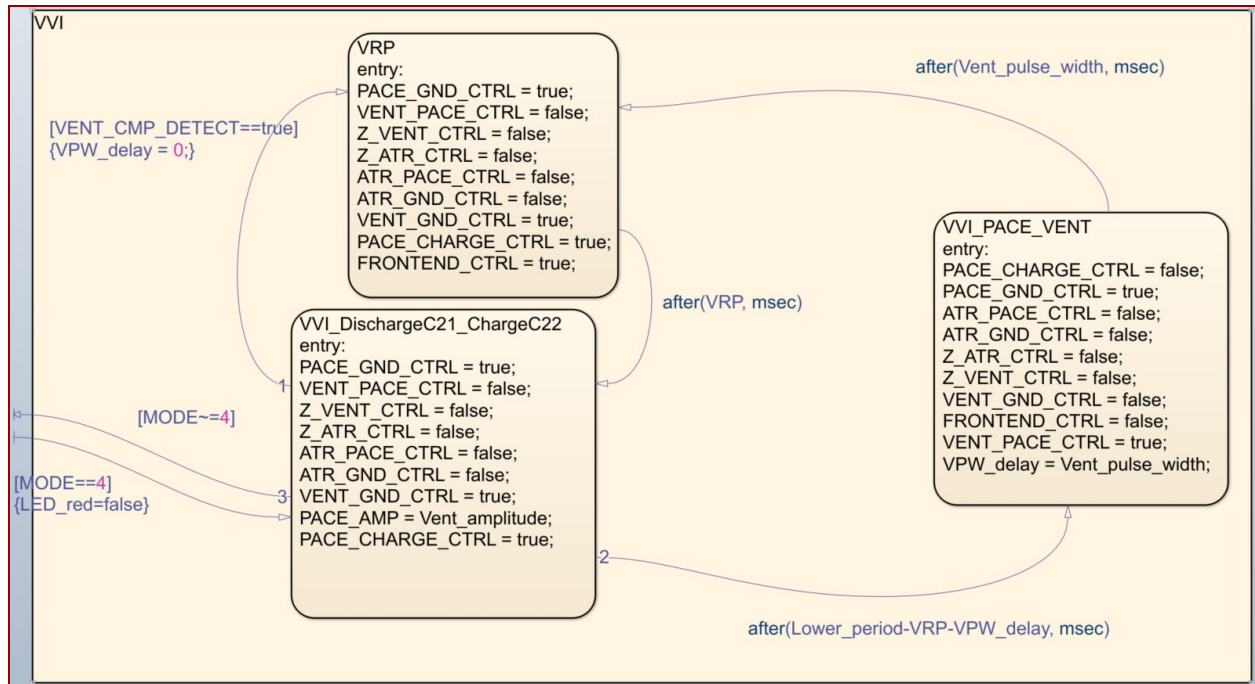
Note that both an artificial pace (from ‘AAI_PACE_ATRIUM’) and a natural pace (from detection while in ‘AAI_DischargeC21_ChargeC22’) trigger the ‘ARP’ state, in which sensing an atrial event does not inhibit the next pace. In the case that an artificial pace is sent, the timing for the following cycle needs to be adjusted to account for waiting the duration of ‘Atrial_pulse_width’, which is why the local stateflow parameter ‘APW_delay’ is set to this duration. Note that this parameter is subtracted from the timing when transitioning from ‘AAI_DischargeC21_ChargeC22’. When a natural pace is detected, this parameter is set to zero since the program was not delayed by a pulse width for that pacing cycle.

The blue LED is activated when a natural pace is detected (see Section 3), and the green LED is activated when ‘ATR_PACE_CTRL’ is true (when an artificial pace is sent). In the pacing state, ‘FRONTEND_CTRL’ is set to false right before setting ‘ATR_PACE_CTRL’ to true, which prevents the

Pacemaker Shield from triggering a sensing signal (and activating the blue LED) based on the artificial pace.

2.5. VVI Stateflow

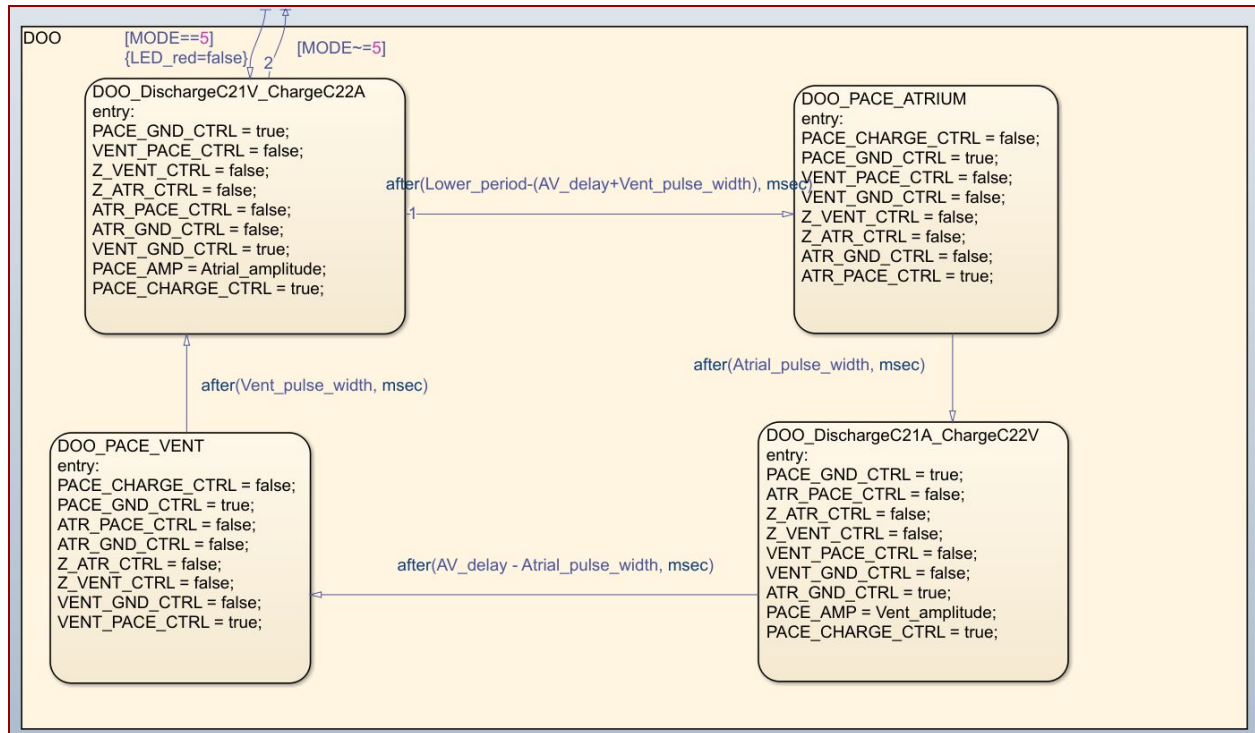
Figure 17: VVI Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 4, the stateflow will enter the VVI set of states shown in Figure 17. The functionality is very similar to AAI, albeit here it is the ventricle being paced and sensed. Thus the states and transitions follow the same logic as the AAI mode, so explanations in Section 2.4 can be used as a reference. Changes are made to pace the ventricle rather than the atrium, to transition based on 'Vent_pulse_width', 'VRP' (for Ventricular Refractory Period) and 'VENT_CMP_DETECT' rather than 'Atrial_pulse_width', 'ARP' and 'ATR_CMP_DETECT', and to and to charge C22 with the voltage of 'Vent_amplitude' rather than 'Atrial_amplitude'.

2.6. DOO Stateflow

Figure 18: DOO Stateflow

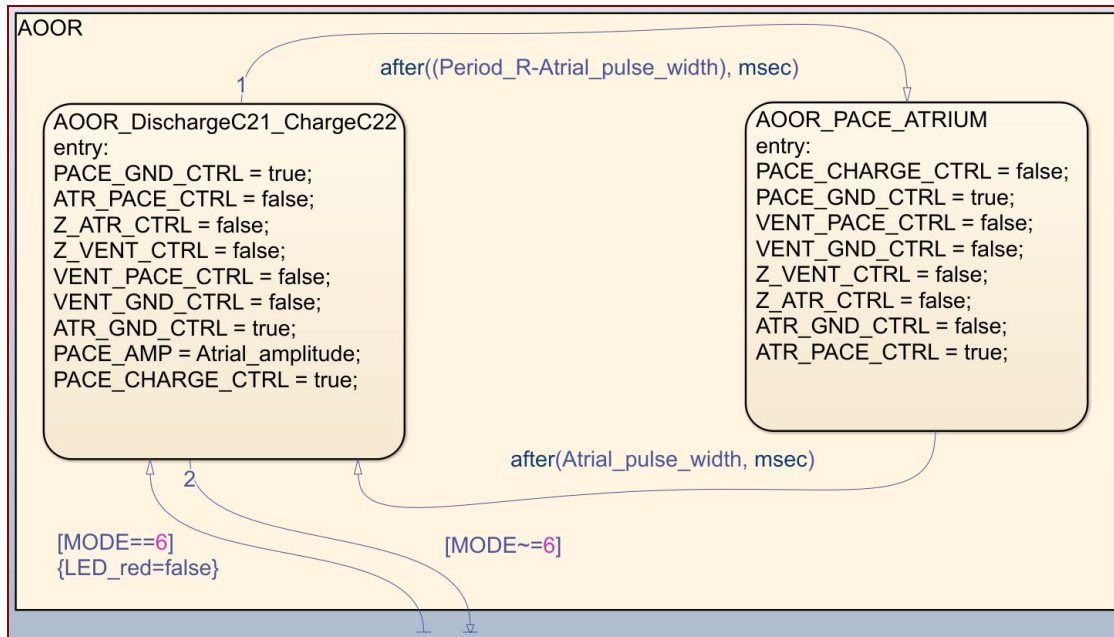


From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 5, the stateflow will enter the DOO set of states shown in Figure 18. Beginning from 'DOO_DischargeC21V_ChargeC22A', this state grounds the preceding ventricular pace (discharges C21 to the ventricle) and charges C22 for an atrium pulse using the voltage of 'Atrial_amplitude'. The program remains in this state for the difference between the desired pacing period and all other durations in the DOO cycle (ie. Lower_period - AV_delay - Vent_pulse_width). 'DOO_PACE_ATRIUM' sets the pins to send an artificial pace to the atrium, in the same way as done in AOO and AAI modes. After the atrial pulse width, 'DOO_DischargeC21A_ChargeC22V' grounds the preceding atrial pace (discharges C21 to the atrium) and charges C22 for a ventricular pace using the voltage of 'Vent_amplitude'. Note that, since the total time between pacing the atrium and the ventricle (ie. time between 'DOO_PACE_ATRIUM' and 'DOO_PACE_VENT') must be equal to 'AV_delay', the time duration for this state must subtract 'Atrial_pulse_width', since that was already incurred before reaching this state. The program then sends an artificial ventricular pace in 'DOO_PACE_VENT', similar to as in VOO and VVI, before returning to 'DOO_DischargeC21V_ChargeC22A'.

As in other modes, the green LED is activated when sending an artificial pace to either the atrium or the ventricle.

2.7. AOOR Stateflow

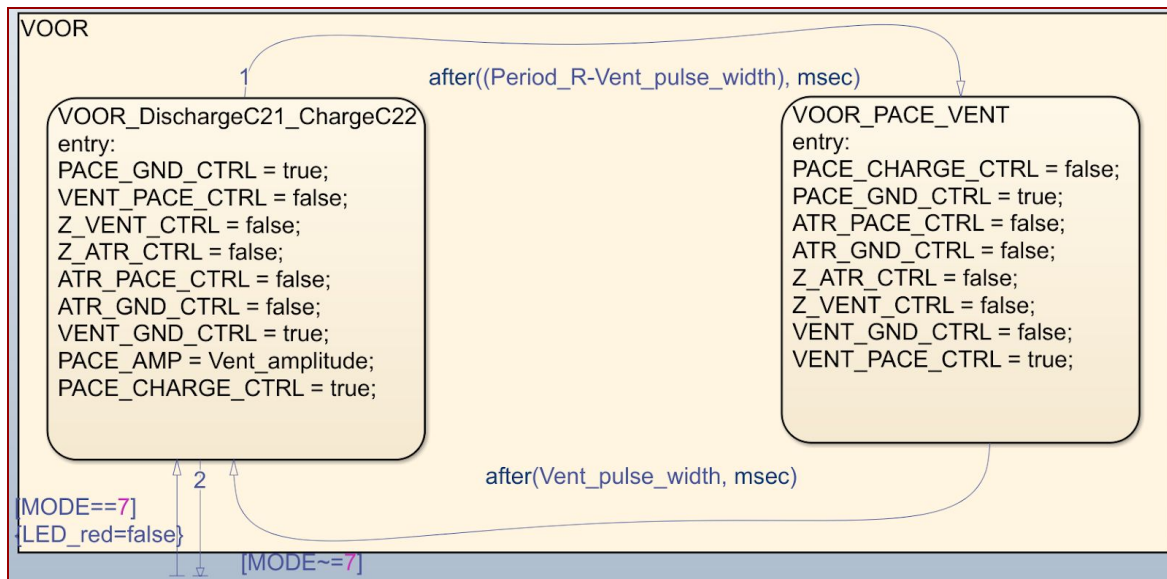
Figure 19: AOOR Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 6, the stateflow will enter the AOOR set of states shown in Figure 19. These states and transitions are identical to those from the AOO mode, but with the important change of using ‘Period_R’ (for a variable-rate period) as the desired pacing period rather than ‘Lower_period’ (fixed-rate period based on LRL). Thus, explanations in Section 2.2 can be used as reference. For more detail on the varying parameter ‘Period_R’, see Section 4.

2.8. VOOR Stateflow

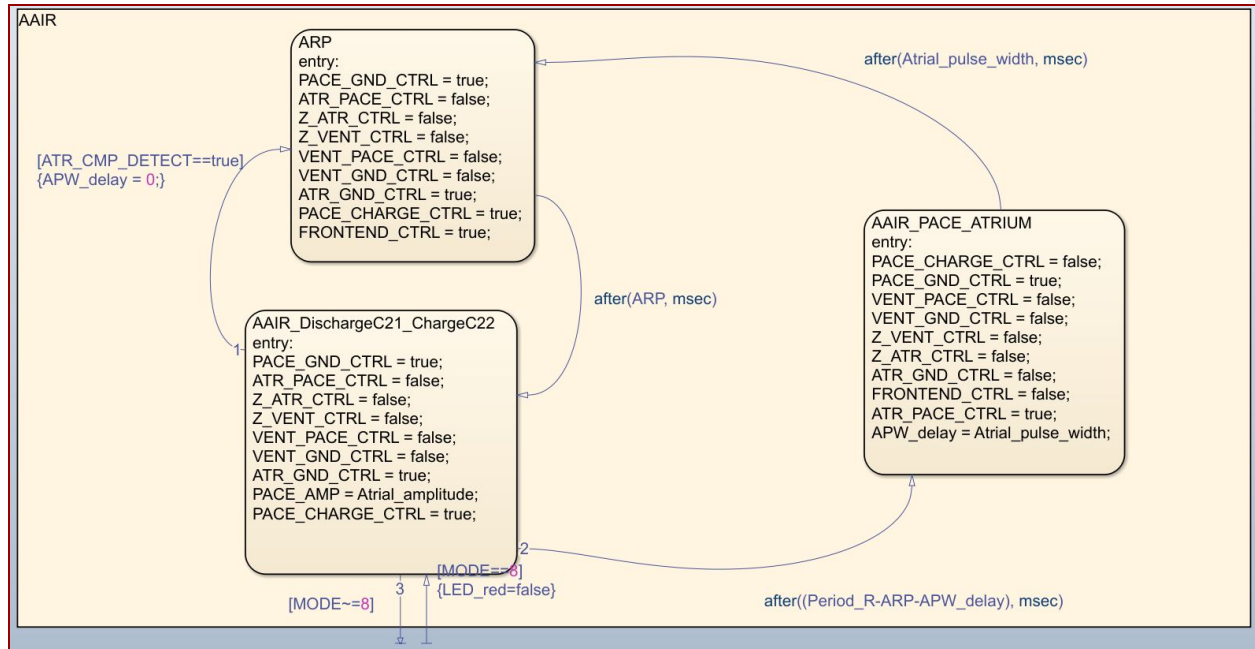
Figure 20: VOOR Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 7, the stateflow will enter the VOOR set of states shown in Figure 20. These states and transitions are identical to those from the VOO mode, but with the important change of using 'Period_R' (for a variable-rate period) as the desired pacing period rather than 'Lower_period' (fixed-rate period based on LRL). Thus, explanations in Section 2.3 can be used as reference. For more detail on the varying parameter 'Period_R', see Section 4.

2.9. AAIR Stateflow

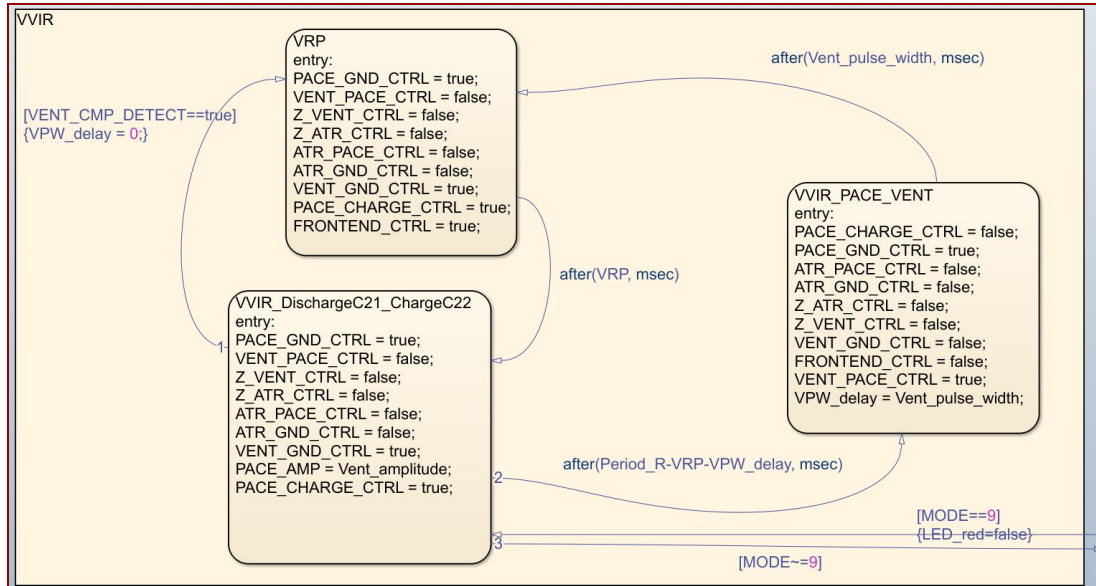
Figure 21: AAIR Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 8, the stateflow will enter the AAIR set of states shown in Figure 21. These states and transitions are identical to those from the AAI mode, but with the important change of using 'Period_R' (for a variable-rate period) as the desired pacing period rather than 'Lower_period' (fixed-rate period based on LRL). Thus, explanations in Section 2.4 can be used as reference. For more detail on the varying parameter 'Period_R', see Section 4.

2.10. VVIR Stateflow

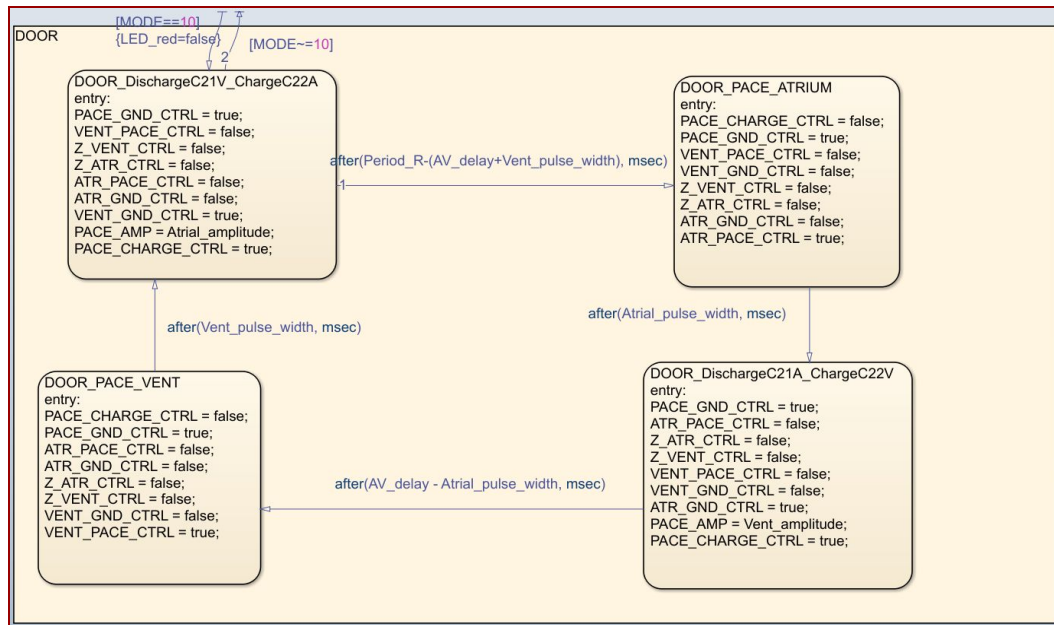
Figure 22: VVIR Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 9, the stateflow will enter the VVIR set of states shown in Figure 22. These states and transitions are identical to those from the VVI mode, but with the important change of using ‘Period_R’ (for a variable-rate period) as the desired pacing period rather than ‘Lower_period’ (fixed-rate period based on LRL). Thus, explanations in Section 2.5 can be used as reference. For more detail on the varying parameter ‘Period_R’, see Section 4.

2.11. DOOR Stateflow

Figure 23: DOOR Stateflow



From the default state, if Lower rate limit is less than Upper rate limit and the MODE is set to 10, the stateflow will enter the DOOR set of states shown in Figure 23. These states and transitions are identical to those from the DOO mode, but with the important change of using 'Period_R' (for a variable-rate period) as the desired pacing period rather than 'Lower_period' (fixed-rate period based on LRL). Thus, explanations in Section 2.6 can be used as reference. For more detail on the varying parameter 'Period_R', see Section 4.

2.12. Design Decisions

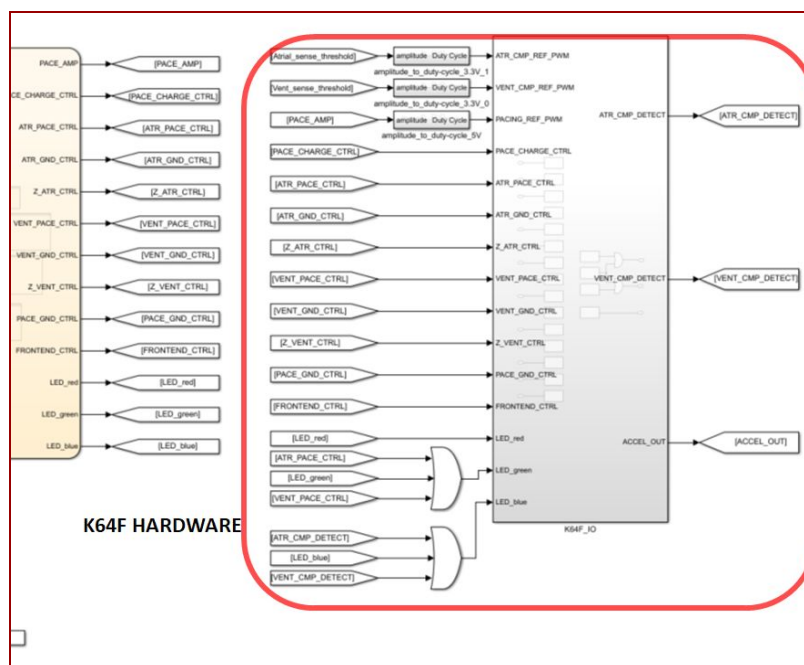
Within the Stateflow chart, subcharted boxes are used to encapsulate the different pacing modes, and no states are shared between modes. This follows the principle of separation of concerns, and leaves each mode as a modular portion that has little dependency on other states (low coupling). Safety verification is also implemented in this program (rather than relying on the DCM) by preventing transitions to pacing modes if the Lower rate limit is greater than the Upper rate limit, or if parameters sent by the DCM are outside of a safe range (see Section 1 for parameter checking). Additionally, in each pacing mode, the K64F's LEDs are used to indicate the state of important signals. The green LED is activated whenever an artificial pace is made by the pacemaker, and the blue LED is activated whenever a natural pace is detected (see Section 3 for details).

2.13. Likely Changes to Requirements and Design

Further development of this project would include requirement changes to implement more advanced pacing modes (such as DDD and DDDR), which would drive the design of additional subcharts within the Stateflow to encapsulate those new modes.

Section 3: K64F Hardware Hiding

Figure 24: K64F Hardware



The goal of hardware hiding is to abstract away the hardware aspect of a system from the perspective of the software - the software is only concerned with the hardware's inputs and outputs, and not how the hardware performs its tasks. As such, if a change is made to the hardware, no changes need to be made to the software, so long as the inputs/outputs remain the same (and vice versa).

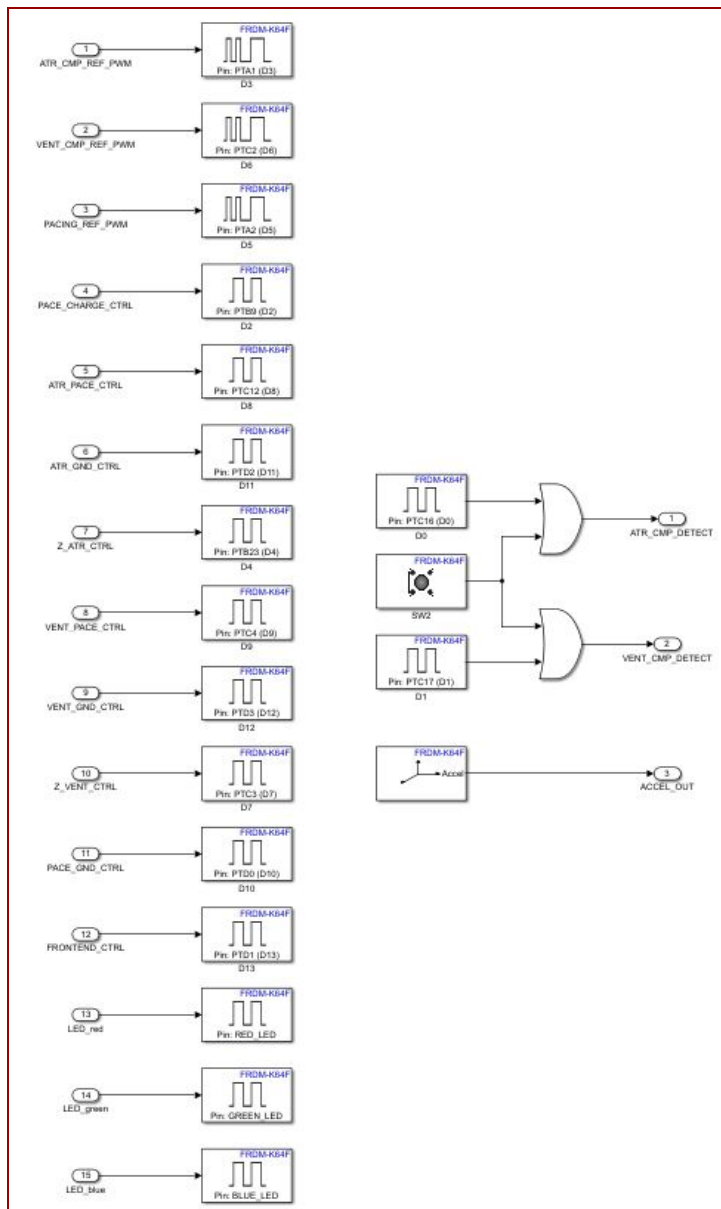
In the simulink model, digital read and write blocks represent the input and output pins of the

microcontroller, which manipulate the pacing and sensing circuitry on the Pacemaker Shield to function as a pacemaker. In order to accomplish hardware hiding, these read/write blocks are placed within a *Subsystem* block (shown in Figure 24), which has predefined inputs and outputs that correspond to the read/write blocks inside. Theoretically, if changes are made to the read/write blocks in how they accomplish their tasks, nothing else in the model needs to be changed since the inputs and outputs remain the same.

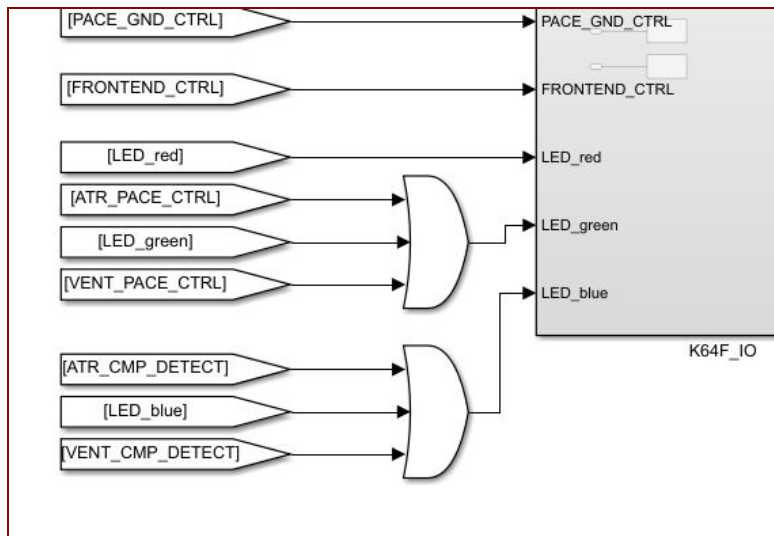
See the Pacemaker Shield documentation for a detailed explanation of each input and output parameter's relation to the pacing and sensing circuitry.

3.1. Inputs

Figure 25: K64F Inputs



Within the Subsystem block, the input blocks are aligned on the left side (shown in Figure 25). The first three write (input) blocks are *PWM Output* blocks, meaning they configure GPIO pins on the microcontroller to send PWM signals to the Pacemaker Shield to charge capacitors. These blocks take input from Duty Cycle values (0-100) for which this model is configured to use type 'uint8'. The duty cycle values are calculated by auxiliary blocks within the model rather than the Stateflow chart (see Section 5). The remaining input blocks are *Digital Write* blocks. These blocks take input from values of type 'boolean', which are determined in various states within the Stateflow chart. The middle 9 blocks configure GPIO pins on the microcontroller to send boolean (high/low) signals to the Pacemaker Shield to open and close switches. The lower 3 blocks control the red, green and blue LEDs on the microcontroller.



Note the addition *OR* gates outside of the subsystem block. These gates allow for activation of LEDs based on boolean values of various important signals, or by parameters in the pacing stateflow. The green LED activates based on three conditions: being set within the pacing stateflow chart ('LED_green'), when an artificial atrial pace is sent ('ATR_PACE_CTRL'), or when an artificial ventricular pace is sent ('VENT_PACE_CTRL'). The blue LED also activates based on three

conditions: being set within the pacing stateflow chart ('LED_blue'), when the hardware detects a natural atrial pulse ('ATR_CMP_DETECT'), or when the hardware detects a natural ventricular pulse ('VENT_CMP_DETECT').

3.2. Outputs

Figure 26: K64F Outputs

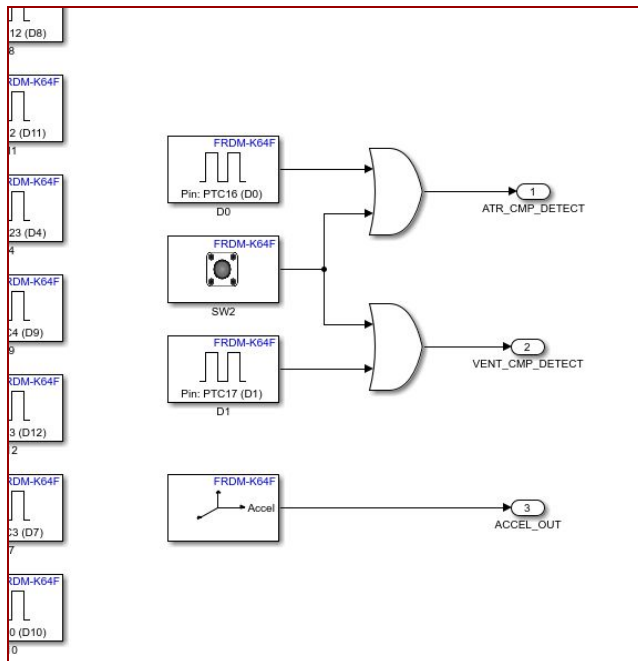


Figure 26 shows the are the read (output) blocks from hardware. There are 2 *Digital Read* blocks, which provide boolean signals corresponding to when the hardware detects an atrial pulse and a ventricular pulse, used by the Stateflow chart. The presence of a *Push Button* block, as well as the *OR* gates, are for the purposes of implementing pace inhibition using the SW2 button on the microcontroller. This configuration causes the pushbutton to trigger the same signal as the detection of a natural atrial or ventricular event.

The accelerometer output block is also present, which allows the model to use raw data from the accelerometer (see Section 4 to see how it is used).

3.3. Design Decisions

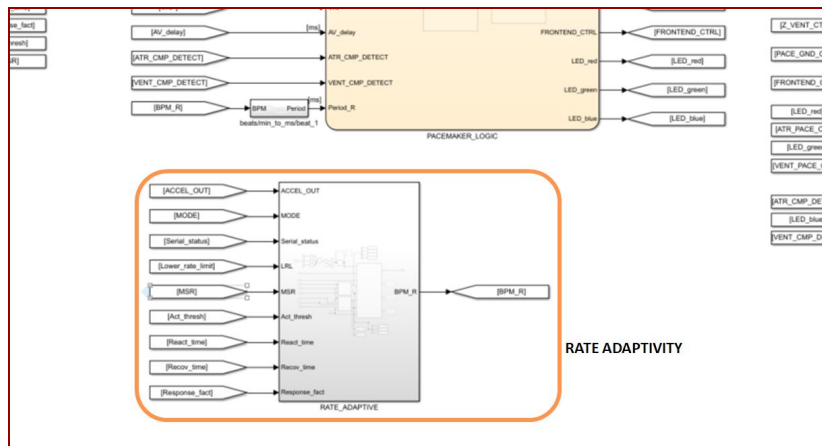
The use of LEDs in hardware provides a useful way to verify and troubleshoot system issues, such as which states the program is cycling through (and a rough measure of their timing), whether natural pulses are being detected properly, and whether critical signals are being sent properly. As present elsewhere in the model, data types are specifically defined for all inputs and outputs of the subsystem block to improve performance.

3.4. Likely Changes to Requirements and Design

Since the requirement changes do not alter hardware and still use the same hardware inputs and outputs, no design changes are required for this section of the model, hence the advantage of hardware hiding.

Section 4: Rate Adaptivity

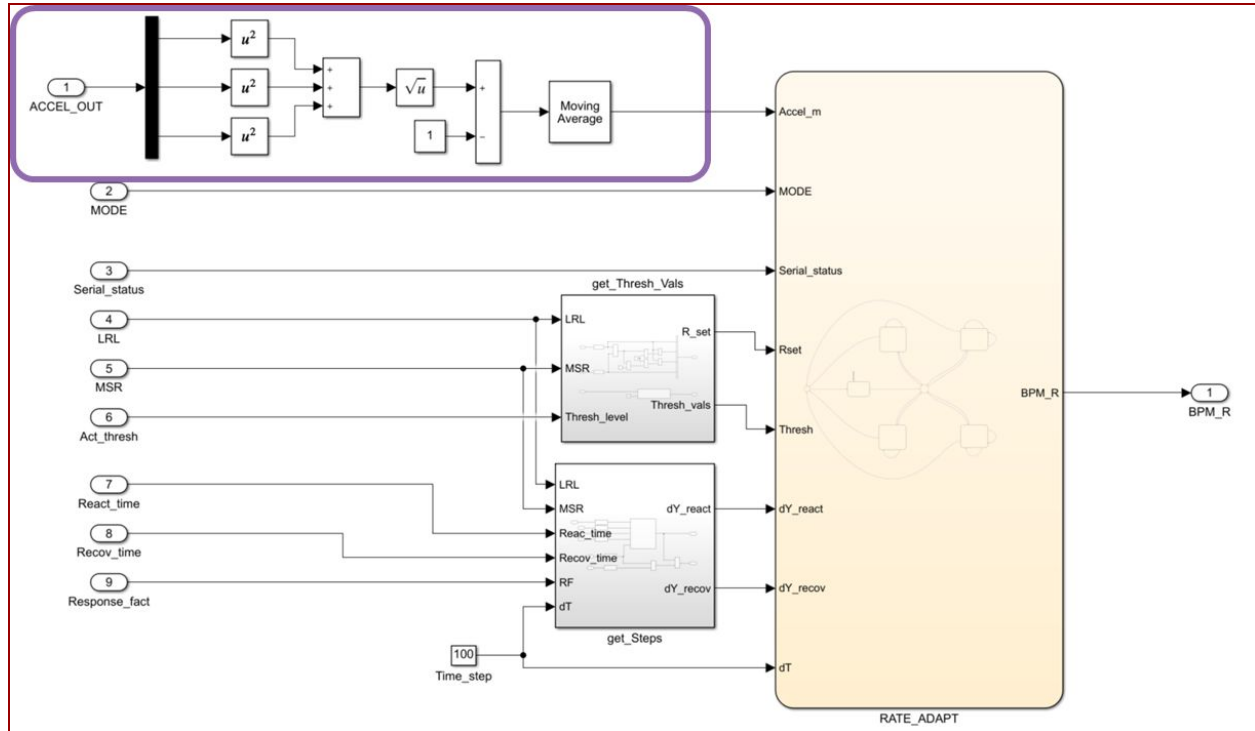
Figure 27: Rate Adaptivity Subsystem (exterior)



The rate adaptivity components are responsible for producing a time-varying rate parameter ('BPM_R') that represents the current rate at which to pace while in rate adaptive modes (ie. XXXR modes). Encapsulated in a *Subsystem* block, a number of related inputs are used, including programmable parameters from the DCM as well as the accelerometer's output data.

4.1. Accelerometer Signal Processing

Figure 28: Accelerometer Signal Processing Components

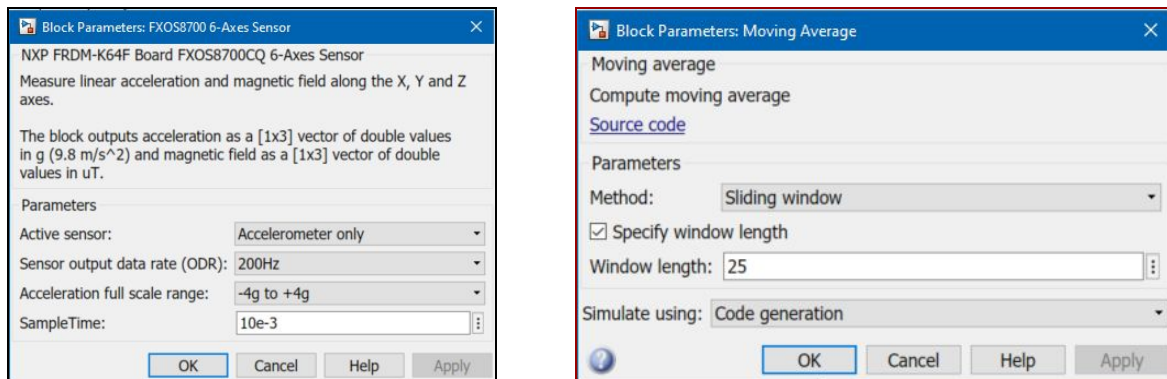


Inside the 'RATE_ADAPTIVE' subsystem, signal processing blocks are enclosed in purple in Figure 28. It can be seen that the raw accelerometer data is processed in the following sequence:

- Decomposed into its X , Y , and Z -axis acceleration data (using a *Multiplexer* block)
- Components are root-sum-squared to find the norm of the acceleration vector (using *Square*, *Add*, and *Square Root* blocks)
- Subtract 1 from the norm, to mitigate (most of) the issue of constantly reading 1g (due to gravity)
- Compute the moving average over a constant window of data points (using a *Moving Average* block)

Extensive experimentation was done to achieve a consistent but representative model of the noisy accelerometer data, while minimizing computational overhead. This experimentation involved tuning the settings of the *Moving Average* and the *6-axis Sensor* block, then viewing the results by running the Simulink model in external mode. The final settings selected are shown in Figure 29a & 29b.

Figure 29a & 29b: Signal Processing Settings



The following three important settings were optimized through iteration:

- Sensor output data rate (how often the sensor outputs a reading): 200Hz (every 5ms)
- SampleTime (time step at which the Simulink program reads the sensor's output): 10ms
- Window length (number of previous inputs used to compute the current average): 25

When tested in external mode using a scope as seen in Figure 30, this method of signal processing produces the reading shown below in Figure 31.

Figure 30: External Mode Testing Configuration

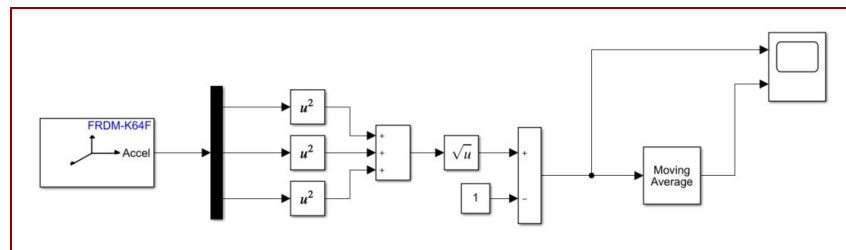
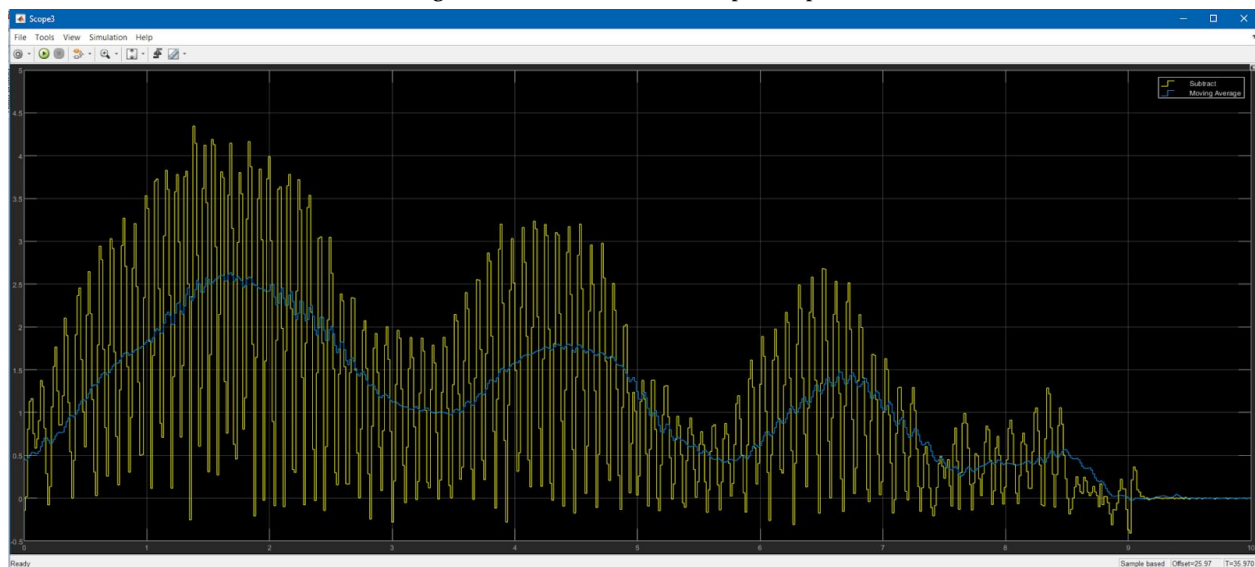


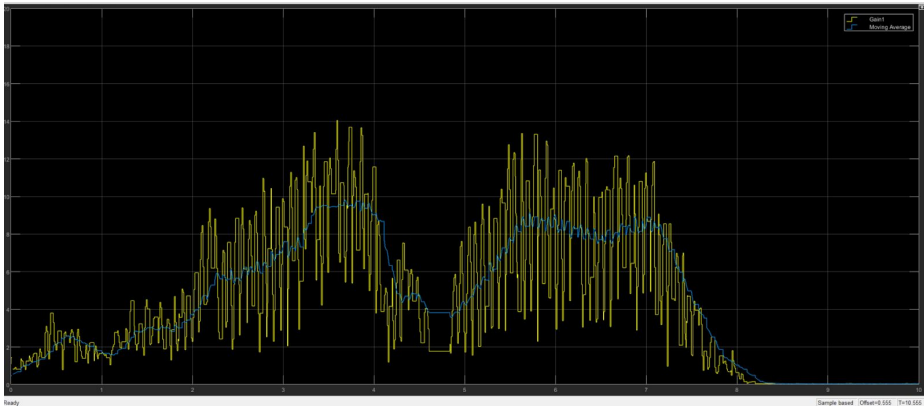
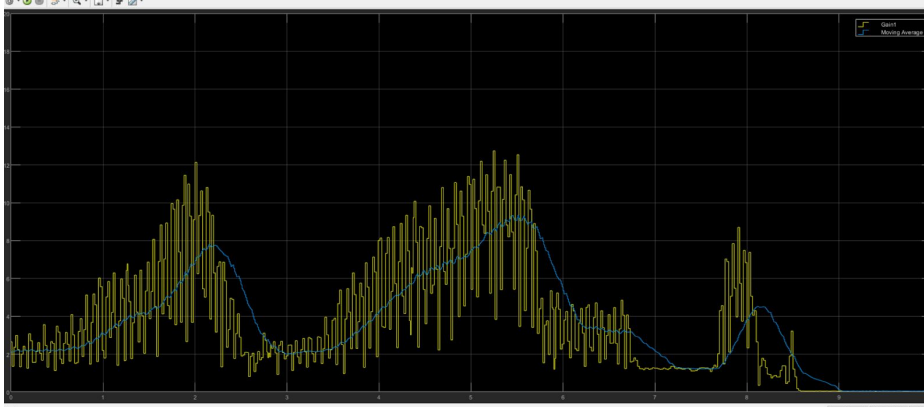
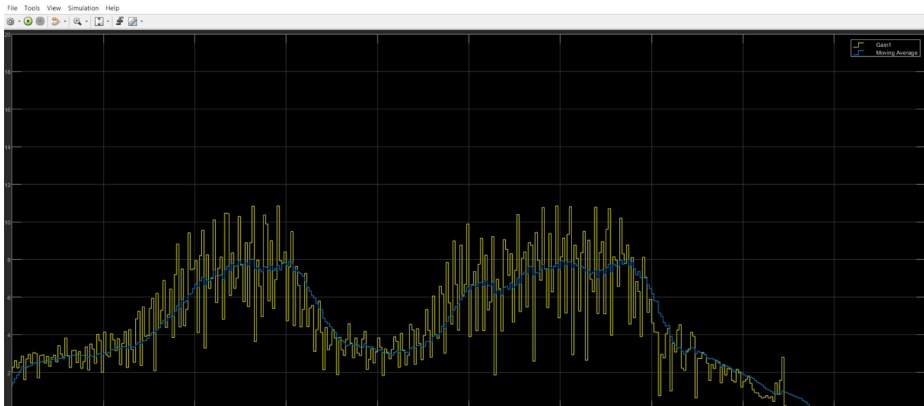
Figure 31: External Mode Scope Output

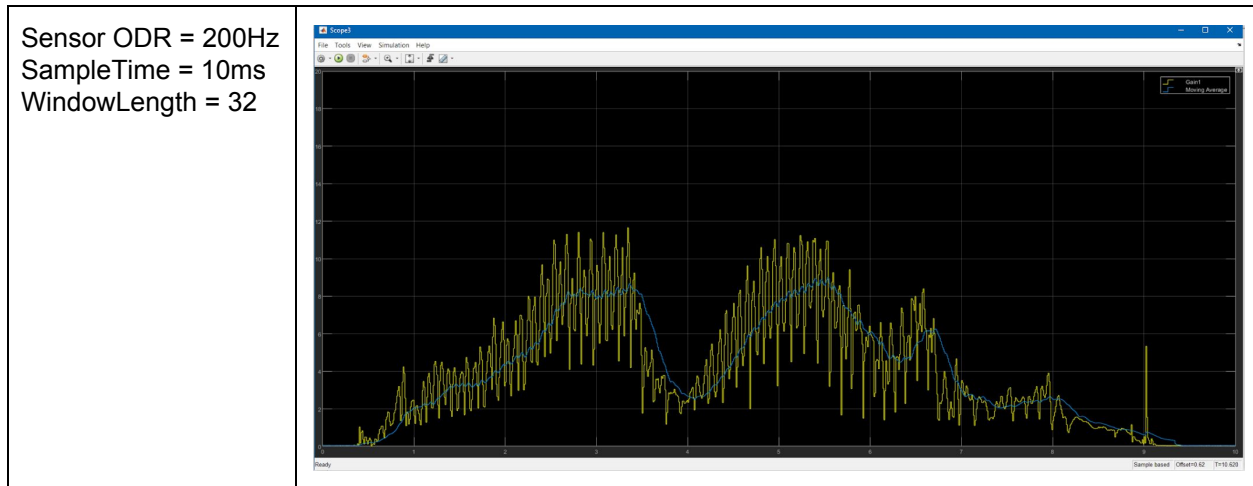


The yellow signal represents the norm of the accelerometer data (after removing the effect of gravity), and the blue signal represents the processed accelerometer signal after the *Moving Average* block. The blue signal is what the program uses to compare to its activity threshold values.

A brief overview of testing with various combinations of these settings is shown in Table 2 below.

Table 2: Signal Processing Testing

Settings	Example Result (as seen by the scope in Figure 30)
Sensor ODR = 100Hz SampleTime = 5ms WindowLength = 50	
Sensor ODR = 50Hz SampleTime = 10ms WindowLength = 50	
Sensor ODR = 100Hz SampleTime = 25ms WindowLength = 12	



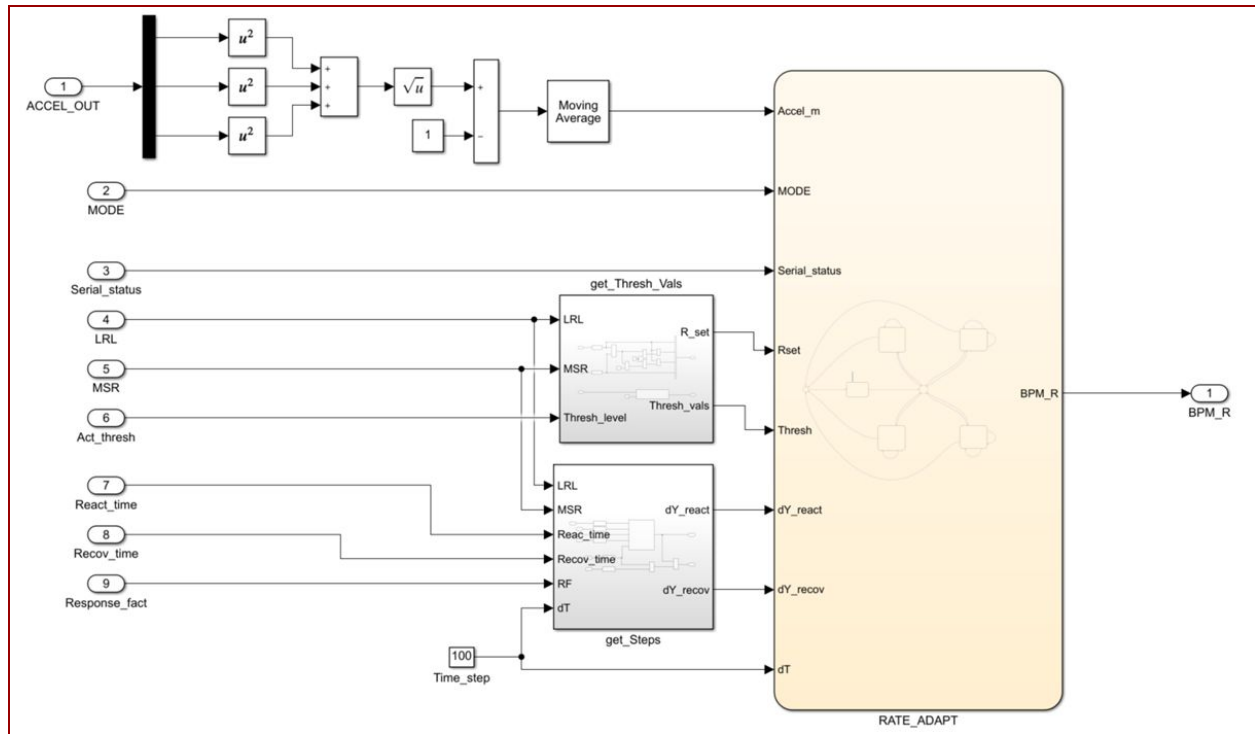
A balance between these settings requires tradeoffs. Increasing WindowLength in general will worsen the effect of “windup” that causes the moving average to output a signal that appears delayed behind the data it is trying to represent. Increasing SampleTime also worsens this effect since the moving average’s window receives data less often and keeps track of data over a larger period of time. For example, in test case 2 from Table 2, the moving average is continually computing the average for the last 500ms of data (10ms SampleTime multiplied by 50 data points), so the blue signal appears delayed behind the accelerometer activity by a large margin. Test case 4 shows the moving average keeping track of 320ms of data, so the delay is less severe but still present. Test case 3 keeps track of 300ms of data, but a small window length of 12 means that the processed signal remains fairly noisy. The final settings selected for the design, resulting in Figure 31, keeps track of 250ms of data to minimize windup while presenting a fairly smooth representation of the accelerometer data. It is also worth noting the final selection of Sensor ODR of 200Hz, meaning that the output data register is loaded with new data every 5ms, and is read at a SampleTime of 10ms. The SampleTime should be equal to (or ideally greater than) the period between ODR loading to avoid reading the same values twice.

If increasing SampleTime worsens the delay of the moving average, why not keep it at 1ms? Through experimentation, it was found that SampleTime values below 5ms (especially at 1ms) had a propensity to cause significant performance losses while running on the K64F, which ruined the precise timing functionality for pacing. It is suspected that the compiled Simulink program encounters issues when trying to read the accelerometer data at a refresh rate comparable to that of the simultaneously-running Stateflow charts, given that the processor is a single-core, single-threaded unit. It was determined that a SampleTime of 10ms or greater would be required to mitigate this issue.

4.2. Varying the Rate Based on Accelerometer Data

Apart from the signal processing components, the remaining blocks in Figure 32 exist solely to produce a linearly varying parameter, ‘BPM_R’, in accordance with the ranges of relevant DCM inputs and the processed accelerometer signal (see Figure 35 for an example). Two *Subsystem* blocks are used to compute variables based on DCM inputs that a Stateflow chart uses to actively monitor and increase/decrease ‘BPM_R’ over a time period.

Figure 32: Rate-varying Components



4.2.1. Computing Threshold Values

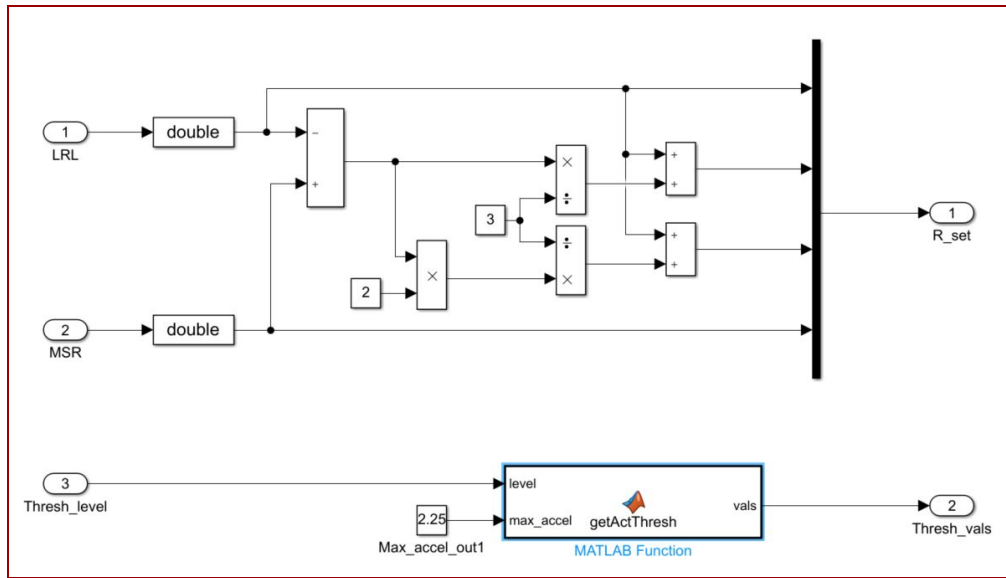
The 'get_Thresh_Vals' subsystem uses the Lower rate limit (BPM), Maximum sensor rate (BPM), and Activity threshold (1-4) to compute a matrix of 4 discrete rate setpoints ('R_set'), as well as a matrix of 3 discrete accelerometer thresholds ("Thresh_vals") that determined which setpoint is active. Each output will be explained in sequence.

The goal of 'Rset' is to create 4 evenly spaced setpoints between LRL and MSR (inclusive). To accomplish this, the following mathematical model was developed:

$$Rset(k) = (MSR - LRL) \frac{k}{3} + LRL \quad \text{where } k = 0, 1, 2, 3$$

For example, given LRL = 60 and MSR = 120, 'R_set' would be computed as [60, 80, 100, 120]. In Simulink this is done using various math operator blocks and a *Multiplexer*, as seen in the upper portion of Figure 33.

Figure 33: get_Thresh_vals



The goal of ‘Thresh_vals’ is to create a range of 3 accelerometer values up to (but not including) a determined maximum sensor output, but this range of values must shift closer to the max output when a higher Activity threshold is selected. In essence, an Activity threshold of 1 (low) should require relatively low accelerometer values to trigger an increased rate setpoint, whereas an Activity threshold of 4 (high) should require relatively high accelerometer values to trigger an increased rate setpoint.

Through testing (such as in Section 4.1), a reasonable maximum sensor output was taken to be 2.25. Higher values could be achieved from the processed accelerometer signal, but would require excessive motion while the board is being probed by an oscilloscope. Since Activity threshold is to be the lowest accelerometer reading that triggers an increased pacing rate, the input range of (1, 2, 3, 4) was mapped to a range of corresponding minimum accelerometer values as (0.4, 0.6, 0.8, 1). With these mapped values, a mathematical model similar to that of ‘R_set’ was developed:

$$Thresh\ vals(k) = (max.\ sensor\ out - AT) \frac{k}{3} + AT \quad \text{where } k = 0, 1, 2$$

And where AT = (0.4, 0.6, 0.8, 1) based on the Activity threshold input. For example, an Activity threshold input of 1 would produce ‘Thresh_vals’ of (0.4, 1.017, 1.633), and an Activity threshold of 4 would produce ‘Thresh_vals’ of (1, 1.417, 1.833).

In Simulink this is done through the use of a *Matlab Function* block, shown in the lower portion of Figure 33 with detail in Figure 34. To see how these thresholds are used in the rate adaptive Stateflow, see Section 4.2.3.

Figure 34: getActThresh Function

```

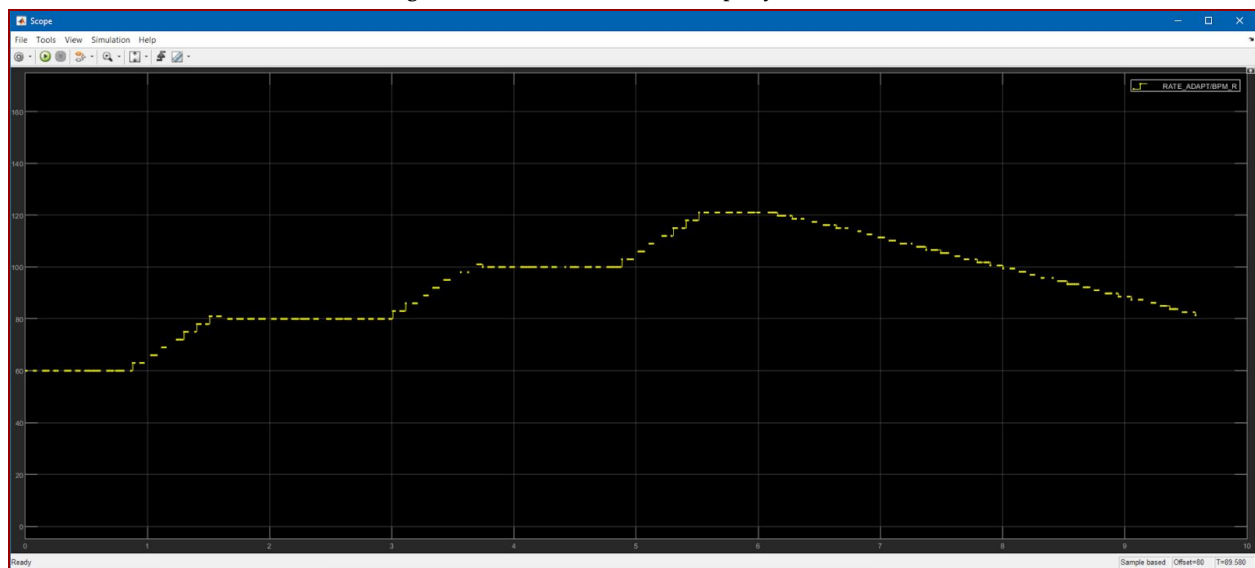
1 function vals = getActThresh(level, max_accel)
2 if level==1
3     AT = 0.4;
4 elseif level==3
5     AT = 0.8;
6 elseif level==4
7     AT = 1.0;
8 else
9     AT = 0.6;
10 end
11 AT1 = (max_accel - AT)*(1/3) + AT;
12 AT2 = (max_accel - AT)*(2/3) + AT;
13 vals = [AT AT1 AT2];

```

4.2.2. Approximating Linear Functions with Discrete Steps

Since the hardware functions in discrete time intervals, following a perfectly linear function over time with infinitely small time steps is impossible. Thus, to approximate a linearly changing variable, a state-machine-like system can be implemented in which, based on the current state, the next state can increment/decrement a variable by a determined amount over a fixed time period. This is the operating principle of this program's rate adaptivity parameter, 'BPM_R'. The 'get_Steps' subsystem shown in Figure 32 computes the necessary increment (reaction) and decrement (recovery) of 'BPM_R' as determined by the current DCM parameters, as well as a predefined fixed time step. To help demonstrate this point, an example of 'BPM_R' varying over time for different accelerometer thresholds is shown below in Figure 35 (taken by attaching a scope to 'BPM_R' and running in external mode).

Figure 35: Discretized Time Steps of BPM_R



To reduce computational complexity, a time step of 100ms is chosen (see Figure 32). A smaller time step would provide higher step resolution but may risk interfering with other time-critical functionality due to the added computational overhead.

According to the provided Boston Scientific documentation, Reaction time is defined as the total time for the rate to increase from LRL to MSR, and similarly, the Recovery time is the time to decrease from MSR to LRL. Simultaneously, the Response factor is supposed to vary the incremental change in rate. Assuming a linear relation with time, this problem is mathematically over-defined since the ΔY (MSR - LRL), ΔT (reaction/recovery time), and the slope (response factor) are all defined. A modification to these definitions must be made to consolidate these parameters. The chosen solution was to find a slope value for both reaction time and recovery time, then to scale this slope using the response factor. Mathematically these are given by:

$$m_{react} = \frac{RF}{8} \left(\frac{MSR - LRL}{T_{react}} \right) \quad m_{recov} = \frac{RF}{8} \left(\frac{MSR - LRL}{T_{recov}} \right)$$

Where RF is the Response factor, T_{react} is the Reaction time, and T_{recov} is the Recovery time. Note that when RF is 8 (its nominal value), T_{react} and T_{recov} fulfill their regular definition. When RF is 16, the slopes are scaled by 2 (largest incremental change in rate), and when RF is 1, the slopes are scaled by $\frac{1}{8}$ (smallest incremental change). Now that the slope values are defined, multiplying by the predefined time step of 100ms gives a step size (ie. $\Delta Y = m \cdot \Delta T$):

$$\Delta Y_{react} = \left[\frac{RF}{8} \left(\frac{MSR - LRL}{T_{react}} \right) \right] \Delta T \quad \Delta Y_{recov} = \left[\frac{RF}{8} \left(\frac{MSR - LRL}{T_{recov}} \right) \right] \Delta T$$

To simplify the calculation and reduce repetitive computation in the program, it can be shown that:

$$\Delta Y_{recov} = \frac{T_{react}}{T_{recov}} \Delta Y_{react}$$

In Simulink, the 'get_Steps' subsystem (see Figure 32) performs these calculations. Details are shown in Figures 36a & 36b:

Figure 36a: get_Steps Subsystem

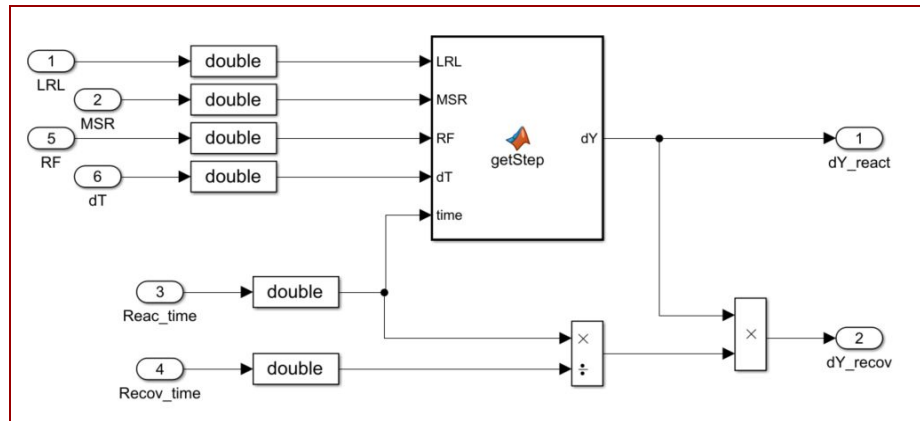


Figure 36b: getStep Matlab Function

```

1  function dY = getStep(LRL, MSR, RF, dT, time)
2  -  dY = ((RF/8) * (MSR-LRL) / (time*1000)) * dT;
3

```

The following plots (Figures 37a & 37b) show how step sizes vary based on differing Response factor values. A Response factor of 16 will have a step size twice that of a Response factor of 8. For both figures, Reaction time is 3s, Recovery time is 10s, LRL is 60bpm and MSR is 120bpm.

Figure 37a: Response Factor of 16

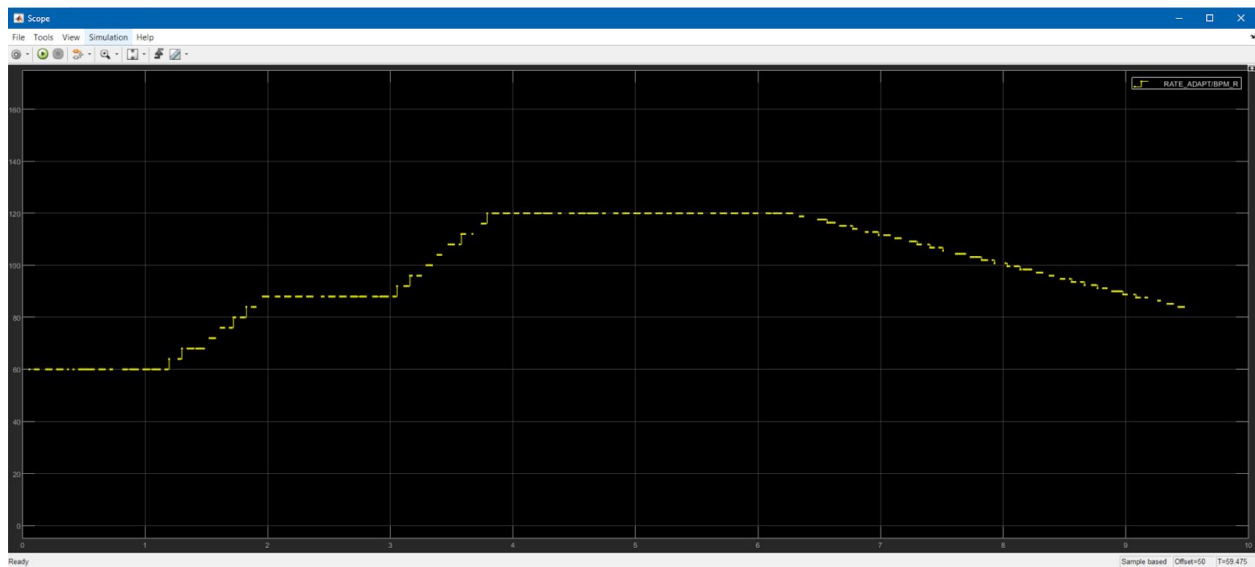
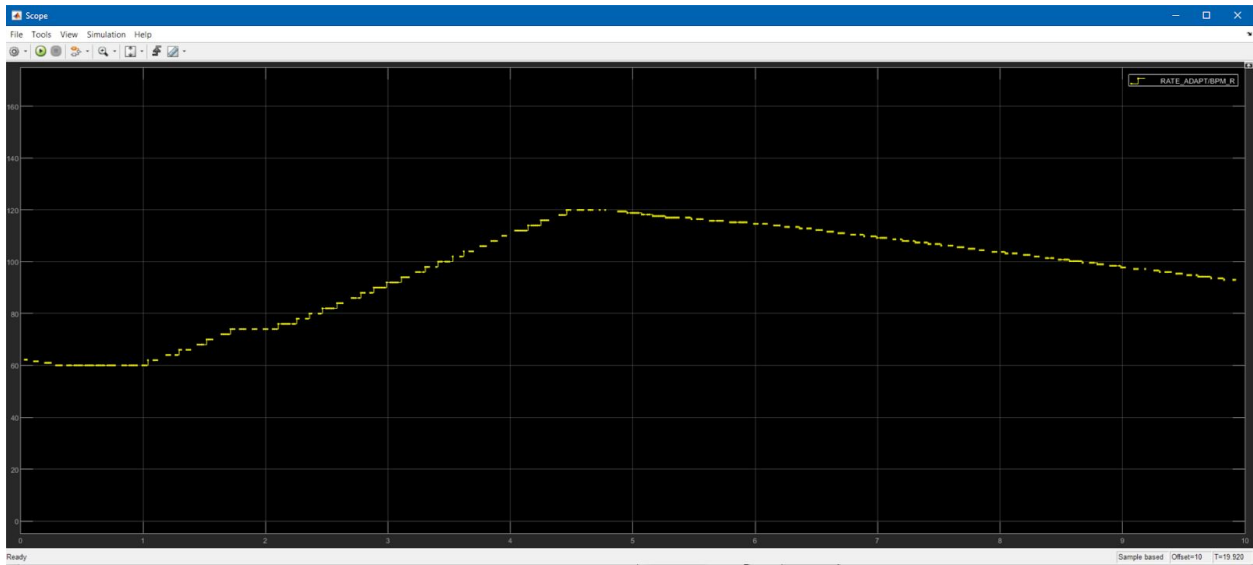


Figure 37b: Response Factor of 8



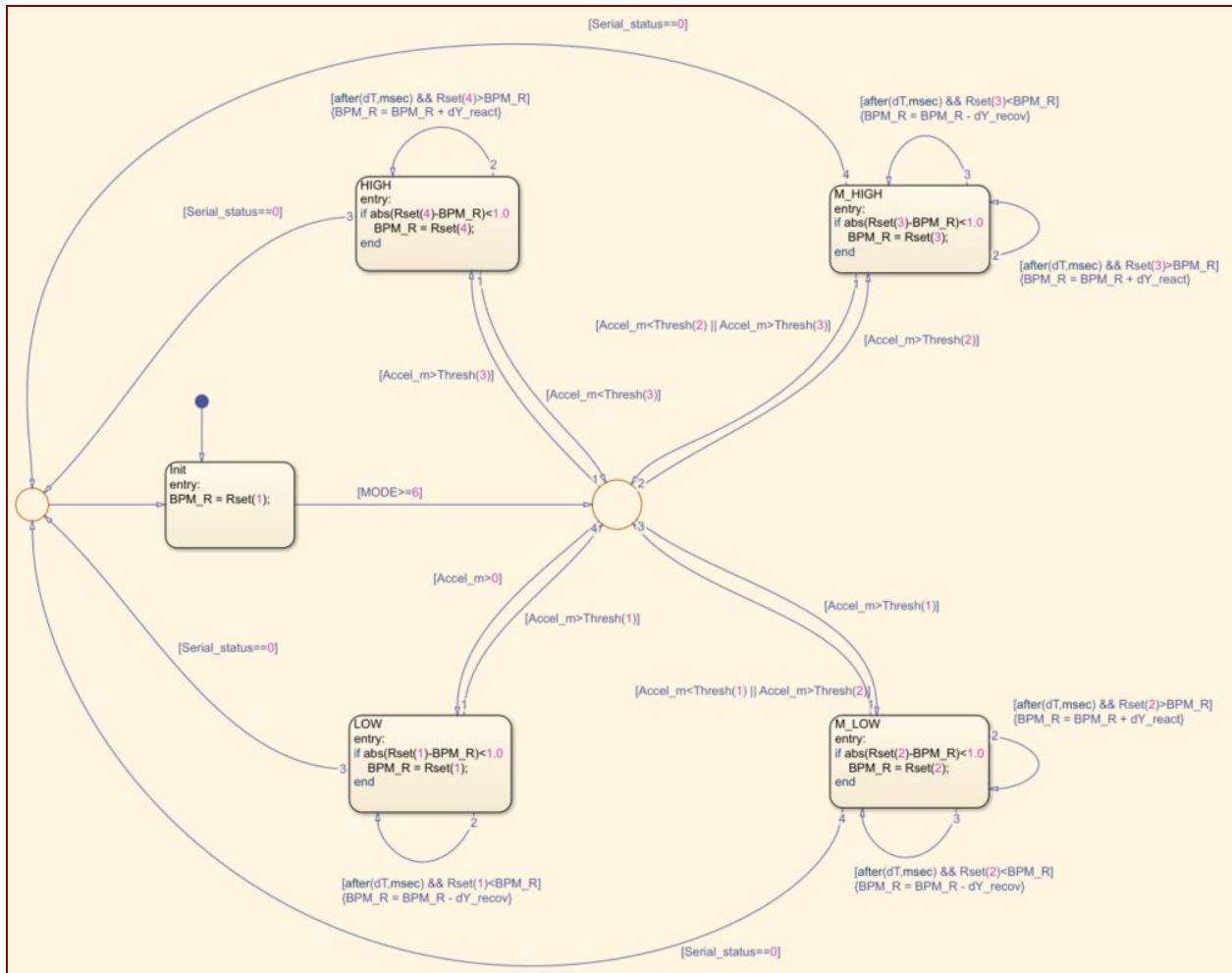
4.2.3. Rate Adaptivity Stateflow

Now that step sizes and time steps are defined (Section 4.2.2), along with threshold values for accelerometer data and rate setpoints (Section 4.2.1), a Stateflow chart is used to increment/decrement 'BPM_R' based on its current value relative to the setpoint at the current activity level. The 'RATE_ADAPT' chart can be seen in Figure 32, and is shown in detail below in Figure 38.

The initial state ('Init') simply sets 'BPM_R' to the first rate setpoint, which is always equal to LRL, before immediately transitioning into one of the activity level states via the centre junction. Note the 'Init' state is only exited if in a rate adaptive mode (ie. 'MODE' is 6 or higher) to save computational effort while in fixed-rate modes. There are four activity level states, separated based on whether the current processed accelerometer reading ('Accel_m') is above an activity threshold (see Section 4.2.1).

- 'LOW' is entered when the activity level is less than the first threshold, 'Thresh(1)'. The activity is negligible and the rate should settle to LRL ('Rset(1)'), if not there already. As such, there is only one transition that changes the rate, decrementing 'BPM_R' by one recovery step size ('dY_recov') after each time step ('dT') that it remains greater than LRL.
- 'M_LOW' is entered when the activity level is above the first threshold, 'Thresh(1)', but below the second threshold, 'Thresh(2)'. The activity is sufficient such that the rate should settle at the second rate setpoint, 'Rset(2)'. If the current rate is less than this setpoint (ie. if 'BPM_R' < 'Rset(2)'), then 'BPM_R' is incremented by one step size ('dY_react') for each time step that this remains true. If the current rate is greater than the setpoint (ie. if 'BPM_R' > 'Rset(2)'), then 'BPM_R' is decremented by one step size ('dY_recov') for each time step that this remains true.

Figure 38: RATE_ADAPT Stateflow



- 'M_HIGH' is entered when the activity level is above the second threshold, 'Thresh(2)', but below the third threshold, 'Thresh(3)'. The activity is sufficient such that the rate should settle at the third rate setpoint, 'Rset(3)'. If the current rate is less than this setpoint (ie. if 'BPM_R' < 'Rset(3)'), then 'BPM_R' is incremented by one step size ('dY_react') for each time step that this remains true. If the current rate is greater than this setpoint (ie. if 'BPM_R' > 'Rset(3)'), then 'BPM_R' is decremented by one step size ('dY_recov') for each time step that this remains true.
- 'HIGH' is entered when the activity level is above the highest threshold, 'Thresh(3)'. The activity is extreme and the rate should rise to MSR ('Rset(4)'). As such, there is only one transition that changes the rate, incrementing 'BPM_R' by one step size ('dY_react') after each time step that it remains less than MSR.

Within each activity level state listed above, if the deviation between the desired setpoint and the current rate is within 1bpm (ie. 'abs(Rset(x) - BPM_R) < 1'), the current rate is set equal to the setpoint. This was implemented to avoid "overshooting" or "undershooting" the setpoint due to the discretized step sizes rarely adding to a number perfectly equal to the setpoints.

Moreover, each of the activity level states will be reset to the 'Init' state whenever 'Serial_status' is 0, indicating that new parameters have been sent by the DCM and the current rate should be reset to the new LRL.

4.3. Design Decisions

The majority of unique design decisions for this program are related to rate adaptivity. Much consideration was given to achieving functional requirements while keeping complex or simultaneous computation to a minimum, to avoid timing issues due to performance impairment. Regarding accelerometer signal processing, it was decided that using the norm of the acceleration vector would be a simple implementation for the purposes of this project, that would minimize the need for complex calculations or filtering while still providing representative data when shaking the microcontroller by hand. Moreover, the iterative testing of signal processing parameters, described in detail in Section 4.1, produced a smooth and representative signal without hindering other timing functionality, mostly due to selecting an increased SampleTime of the *6-axis Sensor* block of 10ms.

Other notable design decisions concern the mathematical modelling of the input domain for rate adaptive parameters, such as determining relevant thresholds (Section 4.2.1) and computing step sizes (Section 4.2.2) for the Stateflow to use, which includes resolving the conflicting definitions of Reaction time, Recovery time and Response factor.. The use of *Subsystem* blocks to perform these calculations follows the principle of separation of concerns by encapsulating their implementation and preventing the Stateflow from having to repeat their function with each refresh.

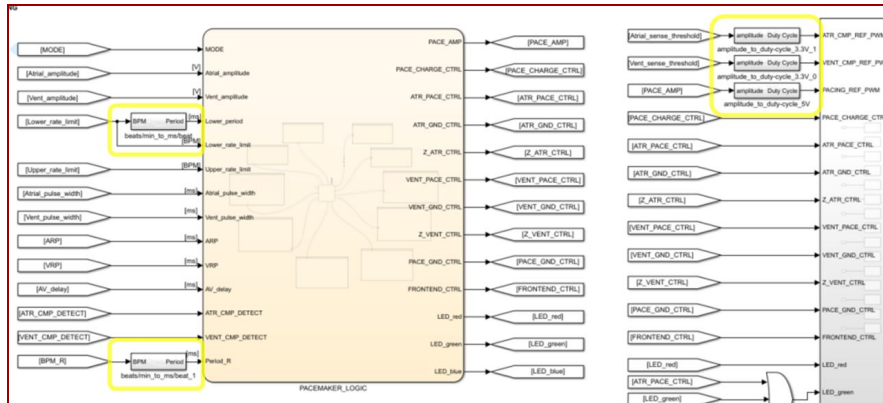
4.4. Likely Changes to Requirements and Design

Further development of this project, with the goal of achieving more realistic rate adaptive functionality, would involve a much more rigorous process for interpreting accelerometer data, such as distinguishing between walking, jogging or running, or riding on a bus with stiff suspension, rather than simply shaking the pacemaker by hand. Moreover, the design decisions made to resolve the rate adaptive parameter conflicts would at least require verification, but would more likely require a complete rework given more coherent requirements.

The current design described by this document already seemed to be pushing the limits of the platform used for this project (K64F hardware, Simulink software). Given the intricate timing requirements of a pacemaker, implementation of more complex functionality would likely require a lower-level development platform that could be much better optimized to the hardware, rather than the high-level abstraction of Simulink.

Section 5: Auxiliary Blocks

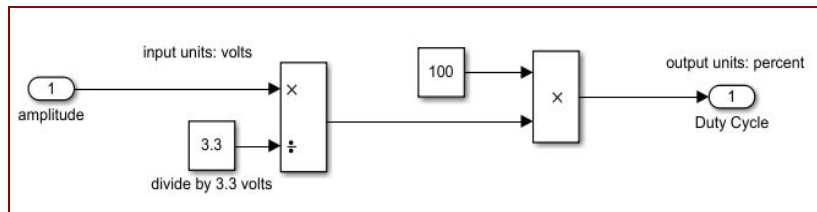
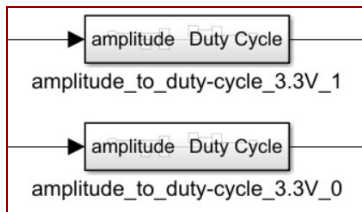
Figure 39: Auxiliary Blocks



The model contains several additional blocks that perform simple unit conversion to simplify calculations done in stateflow logic.

5.1. Amplitude to Duty Cycle (3.3V)

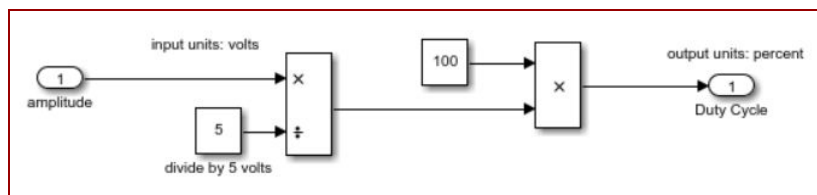
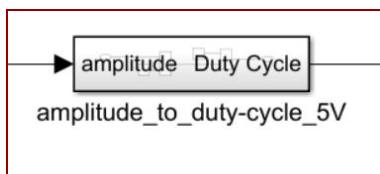
Figures 40a & 40b: Amplitude to Duty Cycle (3.3V)



These blocks take input from ‘Atrial sensing threshold’ and ‘Ventricle sensing threshold’ (see Section 1), and convert their decimal voltages into integer duty cycle values to be passed into the PWM write blocks for the microcontroller (see Section 3). A *Divide* block is used to find the proportion of the input amplitude to the maximum voltage for the capacitor (3.3V), producing a result of type ‘double’. Then a *Product* block is used to multiply this value by 100 and round to an integer number (type ‘uint8’) which corresponds to the duty cycle value.

5.2. Amplitude to Duty Cycle (5V)

Figures 41a & 41b: Amplitude to Duty Cycle (5V)



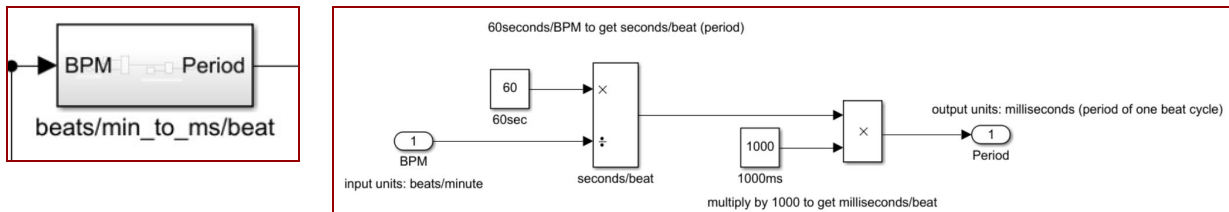
Similar in function to the 3.3V variant, but finds the proportion out of 5V rather than 3.3V, for use in converting ‘Atrial amplitude’ or ‘Ventricular amplitude’ from voltage to duty cycle values. This change is

necessary as the maximum voltage for the C22 pacing capacitor (and thus the maximum pacing amplitude) is 5V rather than 3.3V.

The input to this block is 'PACE_AMP' from the pacing stateflow chart, which is set to 'Atrial amplitude' or 'Ventricular amplitude' depending on whether the program needs to charge C22 for pacing the atrium or the ventricle. The output is written to the 'Digital Write' block (see Section 3) corresponding to the pin that charges C22 using PWM.

5.3. BPM to Period

Figures 42a & 42b: Amplitude to Duty Cycle (3.3V)



This block takes an input value in units of beats per minute, and outputs the corresponding period of one beat cycle (in milliseconds), for use in Stateflow transition logic. One instance of this block converts 'Lower_rate_limit' into 'Lower_period', and the other instance converts 'BPM_R' to 'Period_R'.

5.4. Design Decisions

The purpose of these blocks is to simplify the Stateflow logic. For example, 'BPM to Period' prevents repetitive calculations being done within Stateflow, making the logical transitions simpler and improving performance. The 'Amplitude to Duty Cycle' blocks allow unnecessary signals to bypass the Stateflow entirely by performing the simple unit conversions. All auxiliary blocks follow the principle of separation of concerns, as the services they perform are encapsulated and have limited but well-defined interactions with other sections of the program.

5.5. Likely Changes to Requirements and Design

Since the purpose of these blocks is to encapsulate simple calculations and conversions, other auxiliary blocks can be easily added as needed when implementing more complex features. DDD and DDDR would not require changes to any of the blocks listed in this section.

Appendix A: Blank Simulink Model

