PACEMAKER PARAMETER DOCUMENTATION

JERRY WAN

ZHE WANG

JIANGZHUO HU

CHRIS GEORGE

HAROON JANJUA

Table of Contents

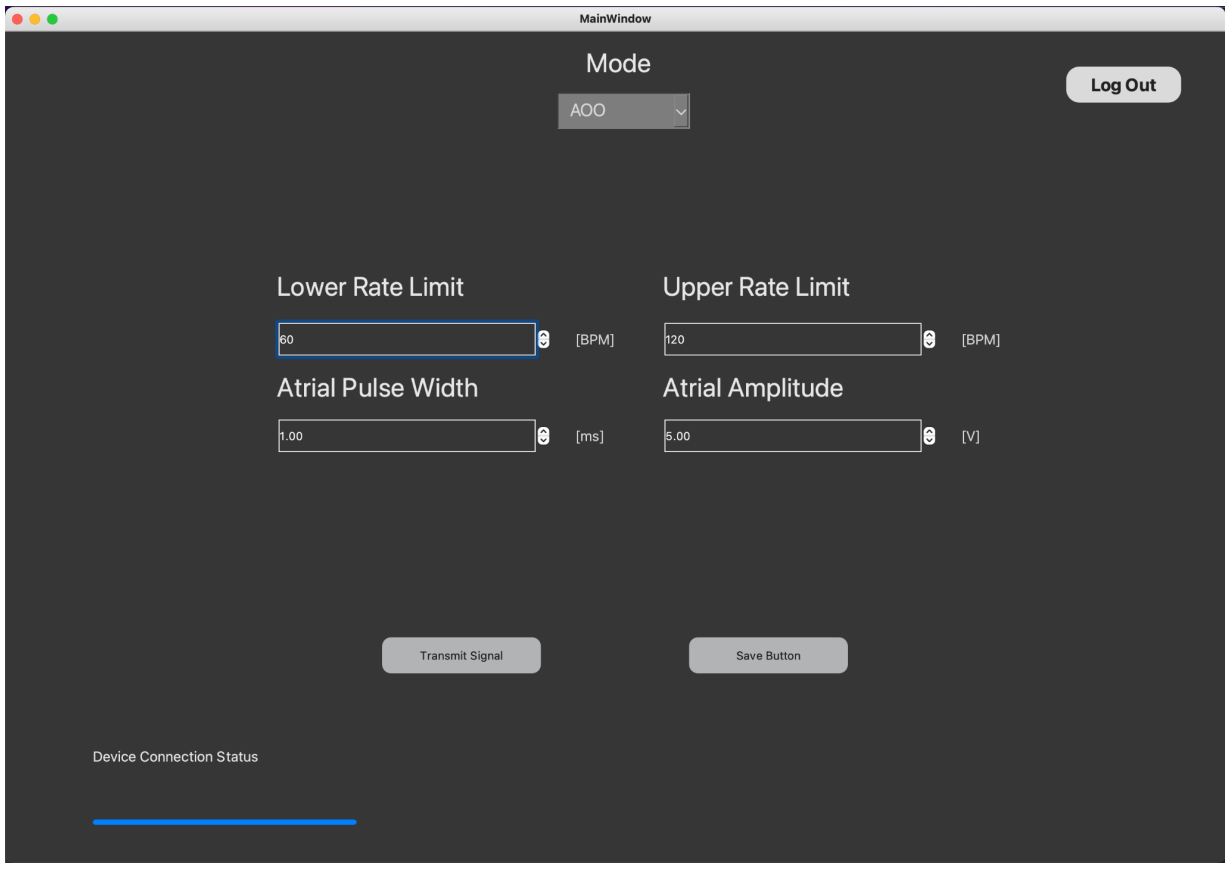
Purpose

The purpose of this documentation is to fulfill section 5 of the document “Assignment 2” found on Avenue, which states: “*Document how programmable parameters originate at the DCM and are implemented in the device. Show how you can ensure the parameters stored in the Pacemaker are what the doctor input on the DCM. Also justify your choice of the data types used to represent parameters data*”.

In that spirit, this document will be broken up into three sections: documenting the origin and implementation of the programmable parameters, a written proof that the parameters in the Pacemaker are the ones in the DCM, and justifying choice of data type for parameter data.

Origination and Implementation of Programmable Parameters

The Pacemaker GUI has four screens, corresponding to four different modes: AOO, AAI, VOO, VVI.



Fig, 1: The AOO Screen

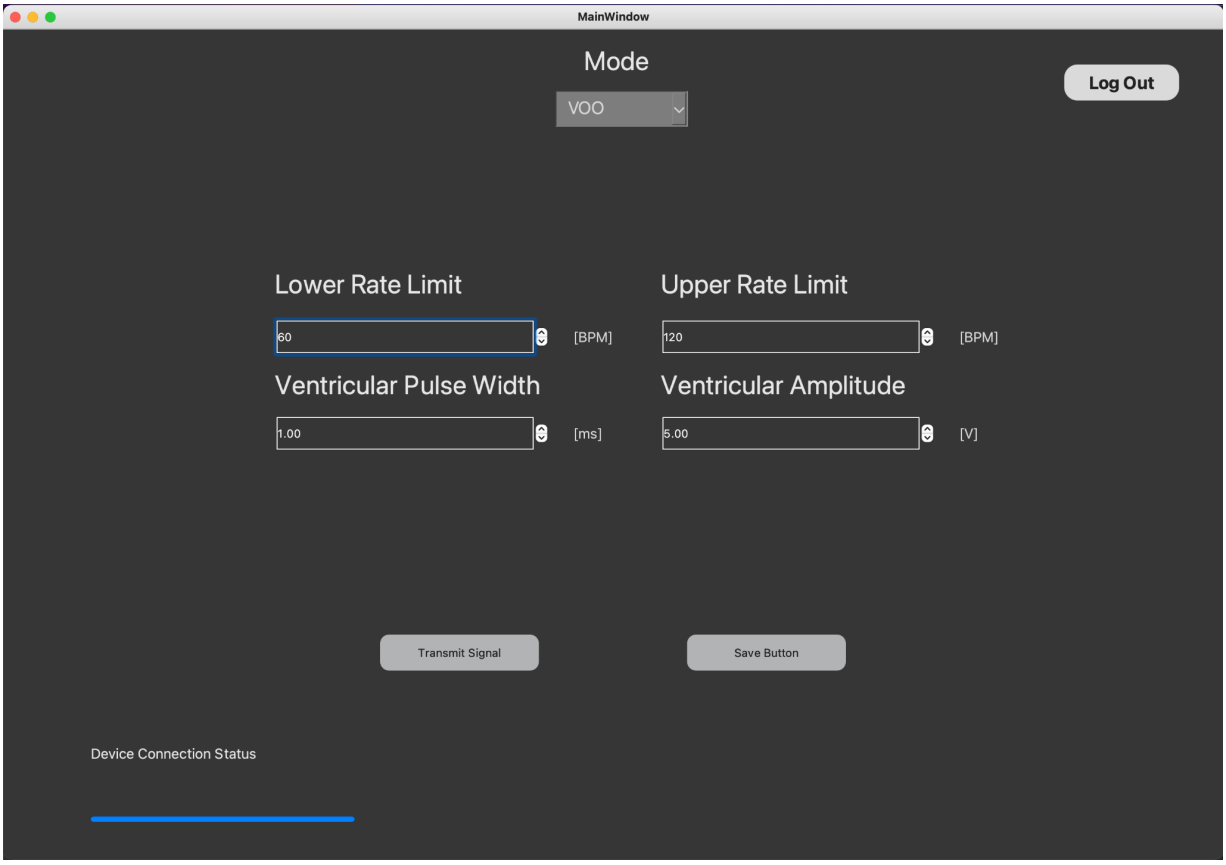


Fig. 2: The VOO Screen

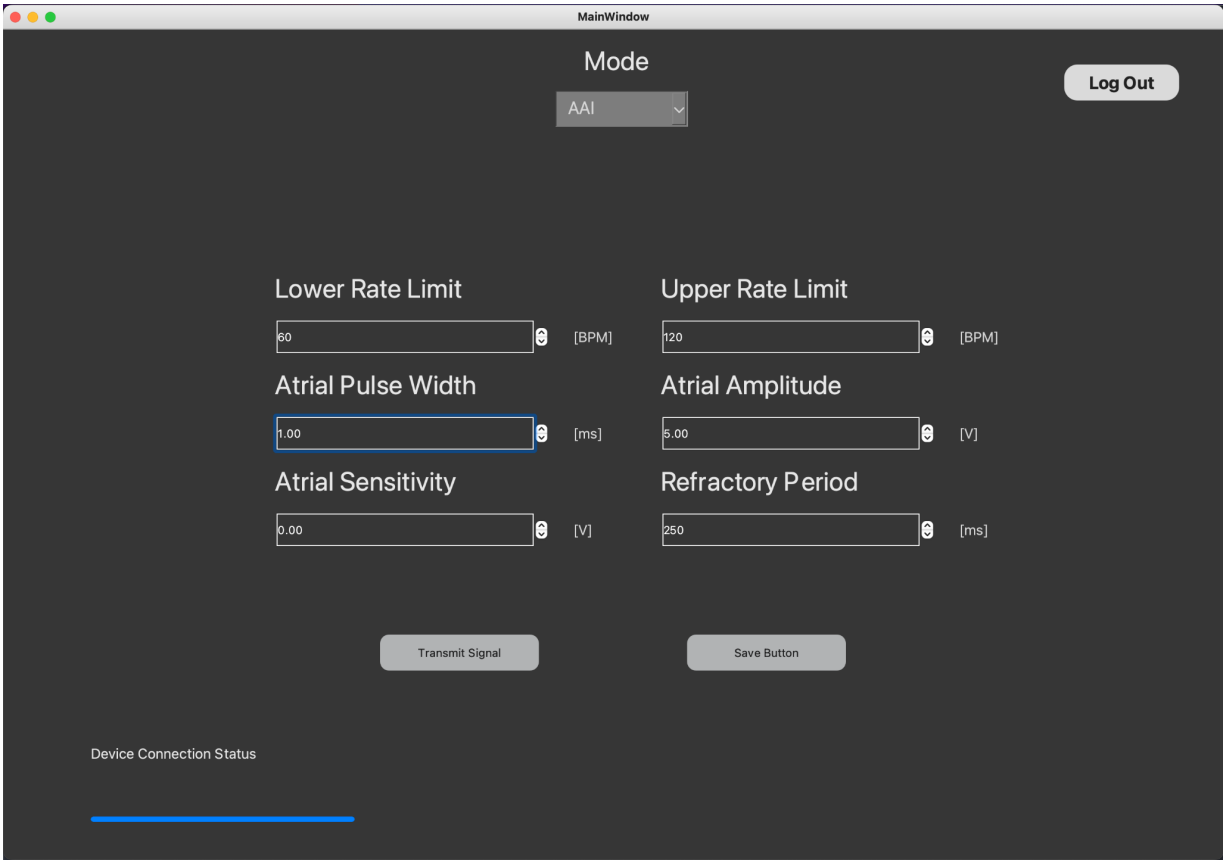


Fig. 3: The AAI Screen

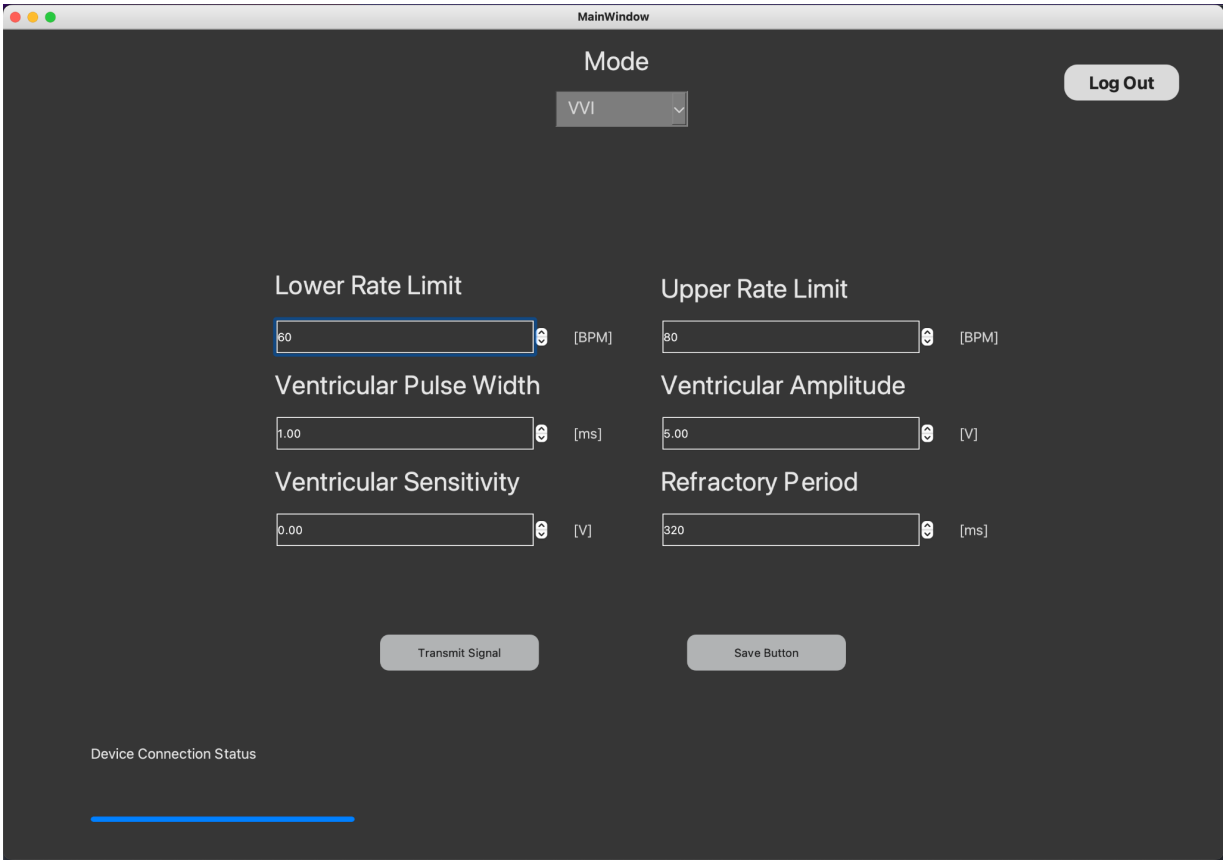


Fig. 4: The VVI Screen

Each mode will take the parameters that are on their specific DCM Screen as input. For ease of reference, here are the parameters taken in by each mode:

Mode	Parameter	Unit
AOO	Lower Rate Limit	Beats Per Minute, or Pulses Per Minute
	Upper Rate Limit	Beats Per Minute, or Pulses Per Minute
	Atrial Pulse Width	Milliseconds
	Atrial Amplitude	Volts

VOO	Lower Rate Limit	Beats Per Minute, or Pulses Per Minute
	Upper Rate Limit	Beats Per Minute, or Pulses Per Minute
	Ventricular Pulse Width	Milliseconds
	Ventricular Amplitude	Volts
AAI	Lower Rate Limit	Beats Per Minute, or Pulses Per Minute
	Upper Rate Limit	Beats Per Minute, or Pulses Per Minute
	Atrial Pulse Width	Milliseconds
	Atrial Amplitude	Volts
	Atrial Sensitivity	Volts
	Refractory Period	Milliseconds
VVI	Lower Rate Limit	Beats Per Minute, or Pulses Per Minute
	Upper Rate Limit	Beats Per Minute, or Pulses Per Minute
	Ventricular Pulse Width	Milliseconds
	Ventricular Amplitude	Volts
	Ventricular Sensitivity	Volts
	Refractory Period	Milliseconds

There are a few functions that initialize and manipulate these parameters.

1. The first is *initializeParameters()*, which is a method of the class *Main* in the file *main.py* under the frontend subdirectory of the Pacemaker interface. It takes the instance of a class as an argument.

2. The next is *update_parameters(),* which is part of the file *db.py* under the database subdirectory of the Pacemaker interface. It takes four arguments: the mode of the Pacemaker, a parameter, a value, and an id.
3. The next is *save_parameters()*, which is a method of the class *Main* in the file *main.py* under the frontend subdirectory of the Pacemaker interface. It takes self as an argument.

Once these functions have manipulated the parameters, they are then sent using PySerial to the Simulink. This is done by implementing the module *struct* and its “pack” and “unpack” methods, and then using PySerial to send parameters to the Simulink side at a baud rate of 115200. Once this is accomplished, the parameters are then used on the simulink side for their given stateflows. After this, the parameters are sent back–this additionally sends the signal outputs from pin A0 and A1, which represent atrial and ventricular electrical activity, respectively. The data from pin A0 and A1 are plotted on the graph when the graph is open and the data from the DCM is transmitted to the Simulink.

Parameters shared between Simulink and DCM

One way to ensure that the values that we transmit is shared between Simulink and DCM is by using the print commands and printing out the unpacked array of data from Simulink and comparing it to the initial values that we send in the DCM. We can also compare the return values to the updated and edited values we transmit in the DCM. Given that logic, we can use the print function and compare it to the values given in the DCM.

Parameters: Data Types and Sizes

The transfer of data from beginning till end can be documented with the following flow:

1. Data is initialized or input on the DCM side
2. Data is then sent to the database to be saved, even upon logout
3. Data is then arranged into an array to be sent to the Simulink Side
4. Data is then used in the input hiding subsystem of the Simulink, and is propagated through the stateflow
5. Data corresponding to the atrial and ventricular electrical activity is then sent back to the DCM side
6. This atrial and ventricular electrical activity is then plotted on the graph window of the DCM.

Therefore, keeping this workflow in mind, we can justify the data type and sizes of our serial transmission by examining the arguments of our pack and unpack functions, and the array of data that we send and receive from Simulink. The array of data we send to our Simulink Model corresponds to:

Parameter	Nominal/Default Value	Max size it can take (bits)
Param 1 (used for Simulink Control)	12	8
Param 2 (used for Simulink Control)	15	8
Pacing Mode	1 (AOO)	8
Atrial Pulse Width	3 [ms]	8

Ventricular Pulse Width	3 [ms]	8
Lower Rate Limit	60 [ppm]	8
Atrial Amplitude	5 [V]	8
Ventricular Amplitude	5 [V]	8
Atrial Refractory Period	250 [ms]	16
Ventricular Refractory Period	320 [ms]	16
Atrial Sensitivity	4 [V]	8
Ventricular Sensitivity	4 [V]	8
Fig. x: the data transmitted and the max size it could be		

When packing our data, we use the struct.pack() documentation to tell us what to cast these values to. From https://docs.python.org/3/library/struct.html, we know that if we are packing data in an array, we can use numbers and data type specifiers to inform the module on what type each value in the array should be cast to. The table shows:

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		(7)
c	char	bytes of length 1	1	
b	signed char	integer	1	(1), (2)
B	unsigned char	integer	1	(2)
?	_Bool	bool	1	(1)
h	short	integer	2	(2)
H	unsigned short	integer	2	(2)
i	int	integer	4	(2)
I	unsigned int	integer	4	(2)
l	long	integer	4	(2)
L	unsigned long	integer	4	(2)
q	long long	integer	8	(2)
Q	unsigned long long	integer	8	(2)
n	ssize_t	integer		(3)
N	size_t	integer		(3)
e	(6)	float	2	(4)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	bytes		(9)
p	char[]	bytes		(8)
P	void*	integer		(5)

Fig Y: the data types in C and their corresponding byte size, Python type, and C type.

From this, we know that the first 8 values of our array should be unsigned chars, with the next two being unsigned shorts, and the final two being unsigned chars. Therefore, we use the argument “8B2H2B” when packing and sending our Parameters to Simulink. This is justified by the fact that there is no more efficient way to store the data.

On the unpacking end, we are receiving the following parameters

Parameter	Nominal/Default Value	Max size it can take (bits)
Readout from pin a0	n/a	64 (double)
Readiut from pin a1	n/a	64 (double)
Pacing Mode	1 (AOO)	8
Atrial Pulse Width	3 [ms]	8
Ventricular Pulse Width	3 [ms]	8
Lower Rate Limit	60 [ppm]	8
Atrial Amplitude	5 [V]	8
Ventricular Amplitude	5 [V]	8
Atrial Refractory Period	250 [ms]	16
Ventricular Refractory Period	320 [ms]	16
Atrial Sensitivity	4 [V]	8
Ventricular Sensitivity	4 [V]	8
Fig. Z: the data transmitted and the max size it could be		

As shown above, the parameters used to control the Simulink Serial Receive have now been overwritten by the readouts of A0 and A1--these pins contain the outputs that we want to plot on our EGrams. These pins contain values that can be best characterized as doubles, and as such they have a greater size and type than the other parameters. For this reason, the unpack argument is given as “2d6B2H2B”–this means that the returned values of the pins A0 and A1 are doubles, and everything else stays the same in data type and size. This, again, is the most compact, efficient, and memory safe way to transmit the data.