Pacemaker Development Assignment 2

Part 2: DCM Design

MECHTRON 3K04: Software Development

TA: Mostafa Ayesh

Fall 2020

2020/11/29

Submitted by: L03 Group 5

Mirza Hassan	**400177534**	**hasanm21@mcmaster.ca**
Anna Hu	**400195637**	**hux57@mcmaster.ca**
Kelvin Huynh	**400179088**	**huynhk12@mcmaster.ca**
Saihaj Gill	**400178643**	**gills69@mcmaster.ca**
Longzhang Zhu	**400199264**	**zhul49@mcmaster.ca**

Revisions

Design Decisions and Requirements of DCM program re-discussed with new implementation of Serial Communication.

Serial Communication Module behavior table added.

Programmable Parameters originating in DCM for use in Pacemaker justified.

Testing of Login/Registration, Database, Menu, and DCM Application modules (and testing for newly implemented modules) moved to Testing document: part3_group5.pdf

Table of Contents

Introduction

A pacemaker device is designed by creating real-time software on hardware platform: NXP FRDM K-64. The DCM plays an integral role in allowing a user to communicate with the hardware. This document will discuss the code developed for the DCM by evaluating likely requirement and design decisions changes, the black box behaviour of each module, internal functions' purpose and behaviour, internal functions' status as public or private, and global variables contained in each module. Further, the implementation of serial communication to transmit and receive information between the DCM and Pacemaker will be described. Overall, this document will provide information about the current status of the DCM and its design. (The testing and functionality of each module can be found in part3_group5 document).

Design Decisions and Requirement Changes

The DCM is designed using the Python coding language. It was determined that using Python was the optimal language for the intended purpose of this software—which is to communicate with the FRDM-K64 hardware board, as the language provides a stable serial communication library. Further, Python provides a suitable environment to create a simple GUI.

The program is made up of 5 interactional modules: login, database, modeInterface, testGUI, and COM. to effectively separate concerns and allow the program to be robust. These modules interact with each other on different levels, and are outlined in the Modules section below.

Many of the requirements have been satisfied, so in terms of requirement changes there are no important changes that need to be made. One requirement change that could be

likely is the addition of extra pacing modes, such as DDDR. Rate adaptive modes have been implemented, but even more advanced pacing modes may need to be added in the future.

Design decisions that may change include re-arrangement of the various interface layouts for the pacemaking modes. In order to use the hardware as an actual pacemaker to be put inside a person, reducing the size of the hardware would be a necessary change.

One thing that needs to be improved is the implementation of the Electrocardiogram. Currently, data can be read from the Electrocardiogram pins (ATR_SIGNAL and VENT_SIGNAL) on the board, however the data that is transmitted does not appear to be correct as it does not match the graphs produced by heartview. One reason for this may be due to a delay in the code execution because of the serial communication, which disrupts pacing and in turn results in incorrect analogue values from the ATR_SIGNAL and VENT_SIGNAL pins to be sent to the DCM.

Furthermore, the efficiency of the serial communications module with the DCM could be improved upon. Currently, the use of PySerial is sufficient, however there is a significant performance issue with the built in read function that reduces the speed at which data is received through the serial port. In addition to this, the DCM is constrained by the need of processing UI elements further reducing the efficiency of serial read operations with regards to the constant Electrocardiogram data polling required. One possible solution would be to introduce multithreading so that the GUI can be run on the main thread simultaneously with the serial read on another. This ensures that the wait time required for the serial read does not impact the execution of code on the GUI side of things.

Modules: Interface Specification and Internal Design
Login/Registration Module (*login.py*)

The login/registration module is designed to interface with the Database module to enable users to register their own accounts and use them to log into the DCM application

Class	Functions	Purpose (a)
Login	__init__(self)	To generate the user interface for logging in
	__validate(self, event)	Retrieves the user input for the username and password fields and validates user credentials
	loggedIn(self)	Accessor method to retrieve state of successful log in
	run(self)	Method to run the Login object and ensures that the gui stays opened until an event occurs
Register	__init__(self)	To generate the user interface for user registration
	__adduser(self, event)	Registers a user provided that the username is unique to the local database containing user credentials
	run(self)	Method to run the Register object and ensures that the gui stays opened until an event occurs
Public Functions (c)	Denoted **WITHOUT** "__" in front	
Black Box Behaviour (d)	Login -> User passes in username and password, receives confirmation of successful or failed login Register -> User passes in username and password, receives confirmation of successful registration or failure to register	
Global Variables (e)	None	
Private Functions (f)	Denoted **WITH** "__" in front	

Database Module (*database.py*)

The Database module is designed to enable storage of login credentials to a locally stored database on the hard-disk.

Functions	Purpose (a)
__init__(self)	Attempts to connect to "storeduserdata.db" file in the same working directory, creates a new database with the same name if database does not exist
createtable(self)	Creates 2 tables if they do not exist in the database. Table 1 (data) stores usernames and passwords Table 2 (params) stores user-specific parameters configured in the menu module
insertuser(self, userdata)	Adds a new user to the database provided a tuple of (username, password). There is a maximum of 10 users. Returns 1 if user was successfully added Returns 0 if user limit is reached
searchusers(self, user)	Searches for username in the database. Returns 1 if the user exists in the database Returns 0 if user does not exist
authenticate(self, user, userinput)	Checks for username and password combination in database against user input. Returns 1 if the user input matches the database Returns 0 if the user input does not match
searchparam(self, user)	Checks for user specific stored parameters from the params table in the database
updateParams(self, parameters)	Takes in parameters in a tuple, and updates the values stored in the params table specific to the current user.

Public Functions (c)	Denoted **WITHOUT** "__" in front
Black Box Behaviour (d)	No user interaction -> No inputs, no outputs explicitly Strictly accessed by login module
Global Variables (e)	None
Private Functions (f)	Denoted **WITH** "__" in front

Menu Module (*modeInterface.py*)

The Menu module consists of the layouts for the various pacemaker modes and allows the editing of the parameter values for the pacemaker. Each mode can be changed into its respective rate adaptive mode by checking the Rate Adaptive option during configuration

Functions	Purpose (a)
__init__(self)	Initialize the pacemaker mode UIs in the form of a notebook object, enabling clean tab selection.
__tabSelection(self, event)	Tab change event detection. Initializes mode interface based on selected tab. Returns mode specific interface function (i.e aooInterface, vooInterface, etc.)
aooInterface(self)	Initializes the UI for the AOO pacemaker mode
__aooInterfaceConfig(self)	Enables a user to manually program the parameters associated with the AOO pacemaker mode
__aooConfigConfirm(self)	Reset UI from programmable to display only for the AOO pacemaker mode
vooInterface(self)	Initializes the UI for the VOO pacemaker mode
__vooInterfaceConfig(self)	Enables a user to manually program the parameters associated with the VOO pacemaker mode
__vooConfigConfirm(self)	Reset UI from programmable to display only for the VOO pacemaker mode
aaiInterface(self)	Initializes the UI for the AAI pacemaker mode
__aaiInterfaceConfig(self)	Enables a user to manually program the parameters associated with the AAI pacemaker mode
__aaiConfigConfirm(self)	Reset UI from programmable to display only for the AAI pacemaker mode
vviInterface(self)	Initializes the UI for the VVI pacemaker mode
__vviInterfaceConfig(self)	Enables a user to manually program the parameters associated with the VVI pacemaker mode
__vviConfigConfirm(self)	Reset UI from programmable to display only for the VVI pacemaker mode

Function	Purpose (a)
dooInterface(self)	Initializes the UI for the DOO pacemaker mode
__dooInterfaceConfig(self)	Enables a user to manually program the parameters associated with the DOO pacemaker mode
__dooConfigConfirm(self)	Reset UI from programmable to display only for the DOO pacemaker mode
commSettings(self)	Initializes the UI for serial communications options including connecting, disconnecting, and receiving echoed parameters
__commConnect(self)	To allow connecting to user-defined serial port
__commDC(self)	To allow closing of serial port
__commSettingsCleanup(self)	To clear widgets used in the commSettings UI for to prevent memory leaking
__commEcho(self)	Enables sending a request to the pacemaker board for it to echo back the currently utilized parameters
__updateValues(self)	Updates list of input parameters to be passed through to the pacemaker. List is structured as: [Mode, LRL, URL, A.amp, A.pw, V.amp, V.pw, VRP, ARP, aSens, vSens, rateAdapt, MSR, actThres, reactTime, resFactor, recTime, avDelay]
validateInput(self, input)	Function used to restrict input of entry fields to numbers and “.”. This enforces values to only be integer or float
egram(self)	Initializes the UI for the egram graphs
__animate(self, i, graph, xvals, yvals)	Graph animation function to update the egram graphs in the egram UI
__startAnimate(self)	Enables button functionality for starting the egram
__stopAnimate(self)	Enables button functionality for stopping the egram
__egramCleanup(self)	Clears the egram UI tab to prevent drawing more than one graph; prevents memory leaking

run(self)	Method to run the Mode object and ensures that the gui stays opened until an event occurs
Public Functions (c)	Denoted **WITHOUT** "__" in front
Black Box Behaviour (d)	Allows user to input and modify various pacemaker parameters, connect to the pacemaker, and monitor egram data.
Global Variables (e)	None
Private Functions (f)	Denoted **WITH** "__" in front

DCM Application (*testGui.py*)

The DCM application is a consolidation of the Login/Registration module and the Menu module intended to create a UI that enables users to control a pacemaker.

Functions	Purpose (a)
__init__(self)	Creates the root window and other GUI elements
login(self)	Creates the login window
register(self)	Creates the user registration window
checkUserLogin(self)	Checks for successful user login
run(self)	Method to run the MainWindow object and ensures that the gui stays opened until an event occurs
Public Functions (c)	Denoted **WITHOUT** "__" in front
Black Box Behaviour (d)	User Interface that can be interacted with
Global Variables (e)	None
Private Functions (f)	Denoted **WITH** "__" in front

Serial Communication Module (*COM.py*)

The Serial Communication handles the serial communication between the UI and the pacemaker, which allows for setting, storing, and transmitting the pacemaker parameters.

Functions	Purpose (a)
__init__(self)	Creates an object of the Serial class and sets the initial connected state to false
serialList(self)	Returns a list of ports that are connected
serOpen(self, serPort)	Takes the specified port (serPort) and attempts to open it, setting the connected state to true upon success
serClose(self)	Closes the serial port and stops communication
serState(self)	Checks if the active port is still open
serialWrite(self, paramList)	Takes a list of parameters, packing the parameters into their respective data types, and transmits it to the board
serialRead(self)	Receives the current parameters that the pacemaker is set to and returns them as a list
startEgram(self)	Requests the egram data from the pacemaker, which is received as a list of doubles to be graphed
stopEgram(self)	Tells the pacemaker to stop transmitting the egram data

Public Functions (c)	Denoted **WITHOUT** "__" in front
Black Box Behaviour (d)	Allows user to open and close serial communication to the pacemaker, read and write parameters as well as receive egram data
Global Variables (e)	None
Private Functions (f)	Denoted **WITH** "__" in front

Serial Communication Programmable Parameters

Parameter	Min	Max	Unit	Data Type	Number of Bytes
sync	-	-	-	uint8	1
FN_code	-	-	-	uint8	1
LRL	30	175	ppm	uint8	1
URL	50	175	ppm	uint8	1
A.amp	0	5	V	single	4
A.pw	1	30	ms	uint8	1
V.amp	0	5	V	single	4
V.pw	1	30	ms	uint8	1
ARP	100	500	ms	uint16	2
VRP	100	500	ms	uint16	2
aSens	0	5	V	single	4
vSens	0	5	V	single	4
Mode	0	4	-	uint8	1
AV Delay	50	300	ms	uint16	2
Activity Threshold	1.1	2.8	-	single	4
Reaction Time	1000	5000	ms	double	8
Recovery Time	600	4800	ms	double	8

MSR	50	175	ppm	uint8	1
Rate Adaptive	0	1	-	uint8	1
Response Factor	1	16	-	uint8	1
				Total	52

Type uint8 is used for parameters sync, FN_code, mode, LRL, URL, A.pw, V.pw, Mode, MSR, Rate Adaptive, and Response Factor because the numerical ranges of these parameters can be measured within the numerical limitation (0 to 255) of 1 uint8 byte. Likewise, type uint16 is used for parameters ARP, VRP and AV Delay as their ranges exceed the numerical limitation of an uint8, and require more bits to represent them which are available from type uint16 (0 to 65 535). Type single is used for parameters A.amp, V.amp, aSens, vSens and Activity Threshold to accurately store their numerical values that may have digits to the right of a decimal. In a similar manner, type double is used for Reaction Time, and Recovery Time for the same reason as a single, but allows for a larger range of numbers. Therefore, a total of 52 bytes are used to store the data between the DCM and Pacemaker.

These programmable parameters originate at the DCM, and are implemented onto a Simulink model under the same date type and byte sizes. This implementation allows for the DCM and hardware to serially communicate with each other.

The parameters stored in the Pacemaker can be checked by simply requesting the parameters from the board and reading them using serial communication. Comparing the values received from the Pacemaker to the values entered by the doctor (on the DCM) would effectively ensure that the Pacemaker is receiving the correct data.

Electrocardiogram

On the pacemaker board there are two pins, ART_Signal and VENT_Signal, which are connected to the analog electrocardiogram signals for each chamber. The data from each pin can be read and converted into a double data type and transmitted using serial communication. The data can then be plotted to create a graph that looks similar to the ones seen in heartview.

The data that was received from the pins did not appear to match the shape of other electrocardiograms, suggesting that there may have been an issue with its implementation. One reason for this could be that way the data is being read may not be done properly. Perhaps there is a better way to obtain the data from the pins. Another reason could be that there is an issue with the serial communication, specifically for the electrocardiogram data. There may be issues with the way the data transmission was implemented in Simulink.

Graphical User Interface Overview

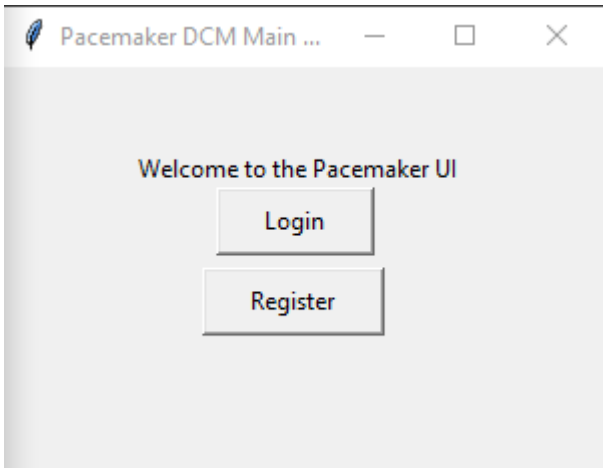

Figure 2: Welcome pop-up window

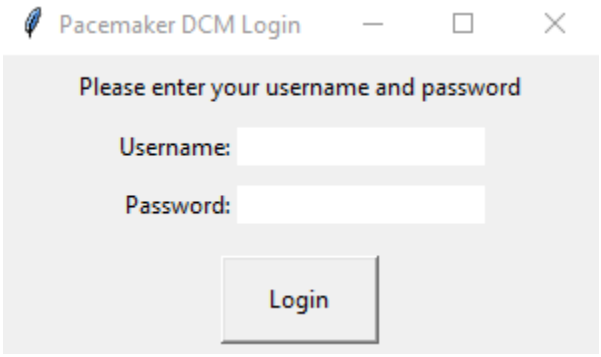

Figure 3: Login pop-up window

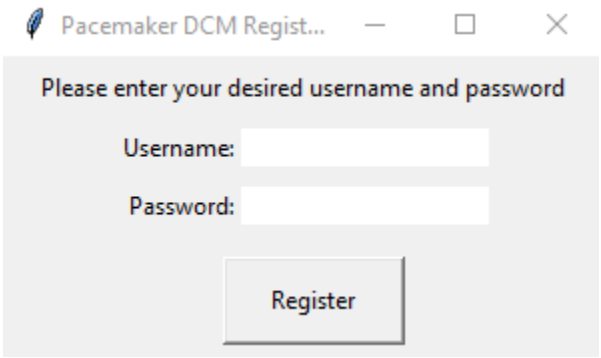

Figure 4: Registration pop-up window

Figure 5: AOO(R) Interface

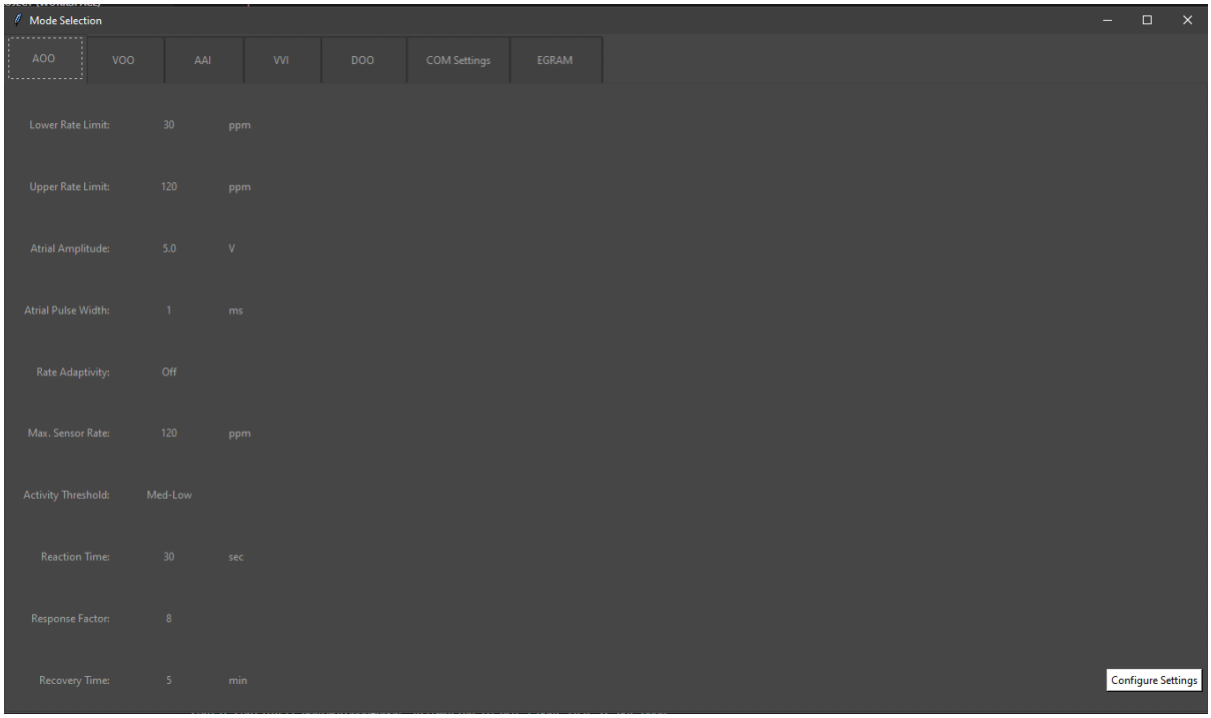

Figure 6: AOO(R) Configuration Menu

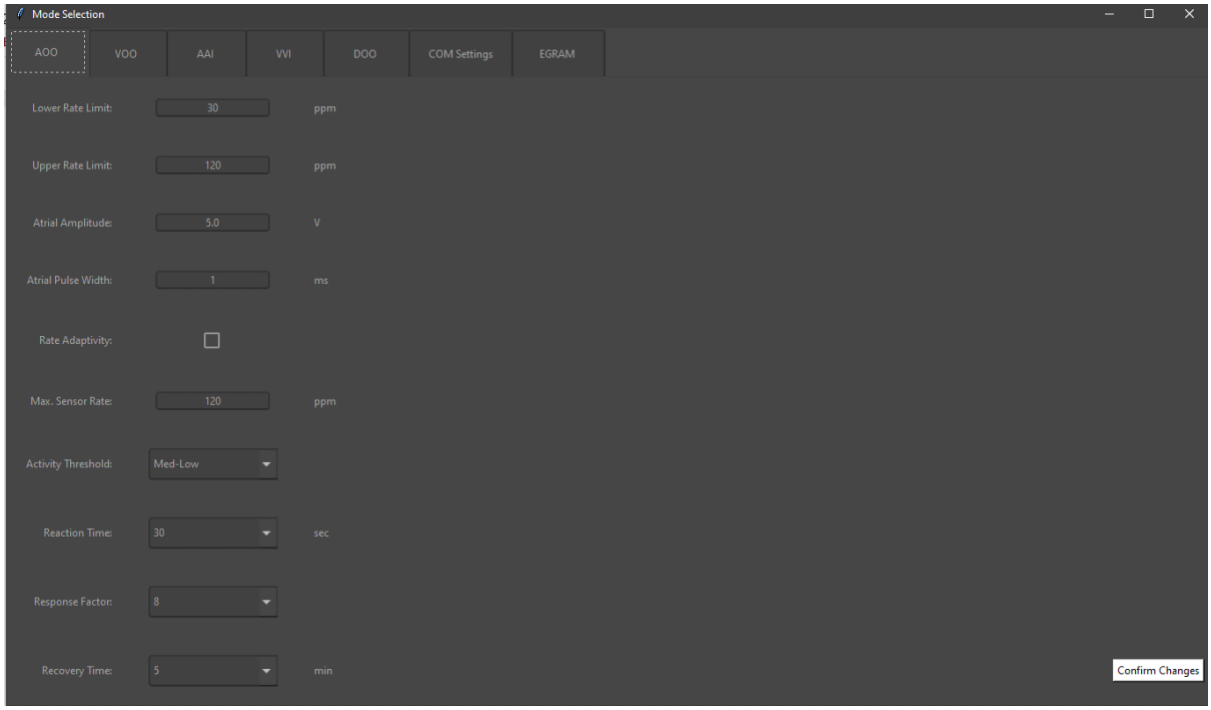

Figure 7: VOO(R) Interface

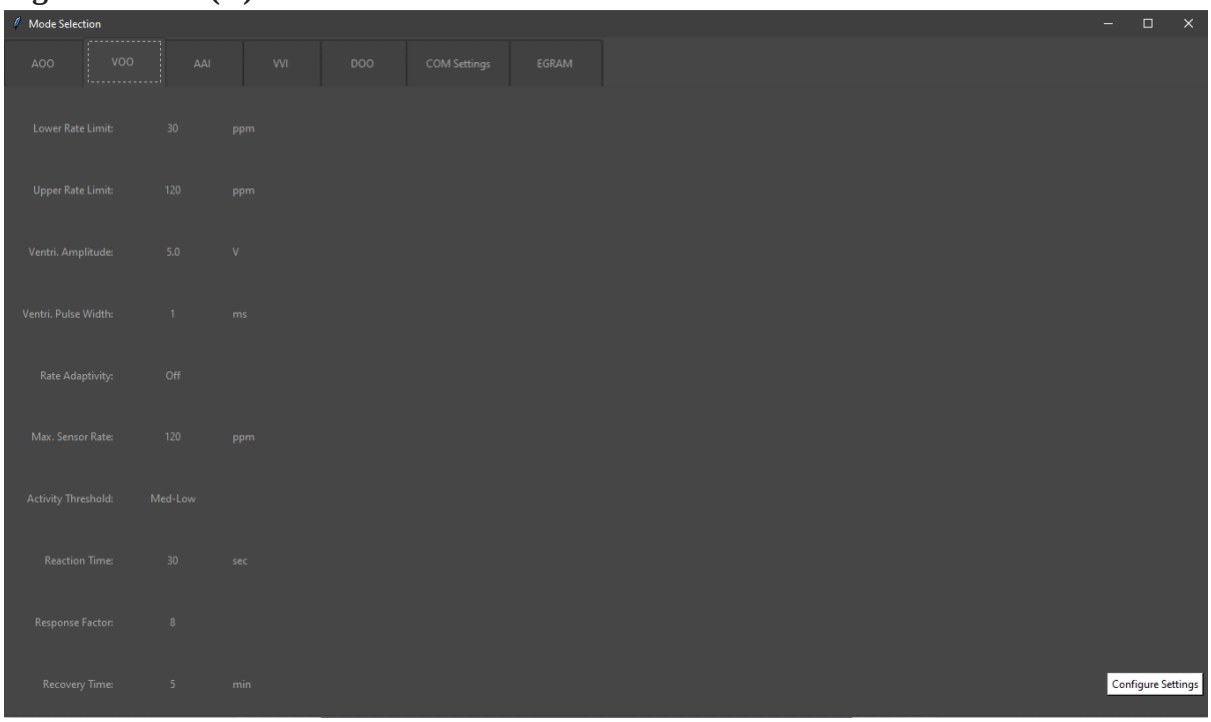

Figure 8: VOO(R) Configuration Menu

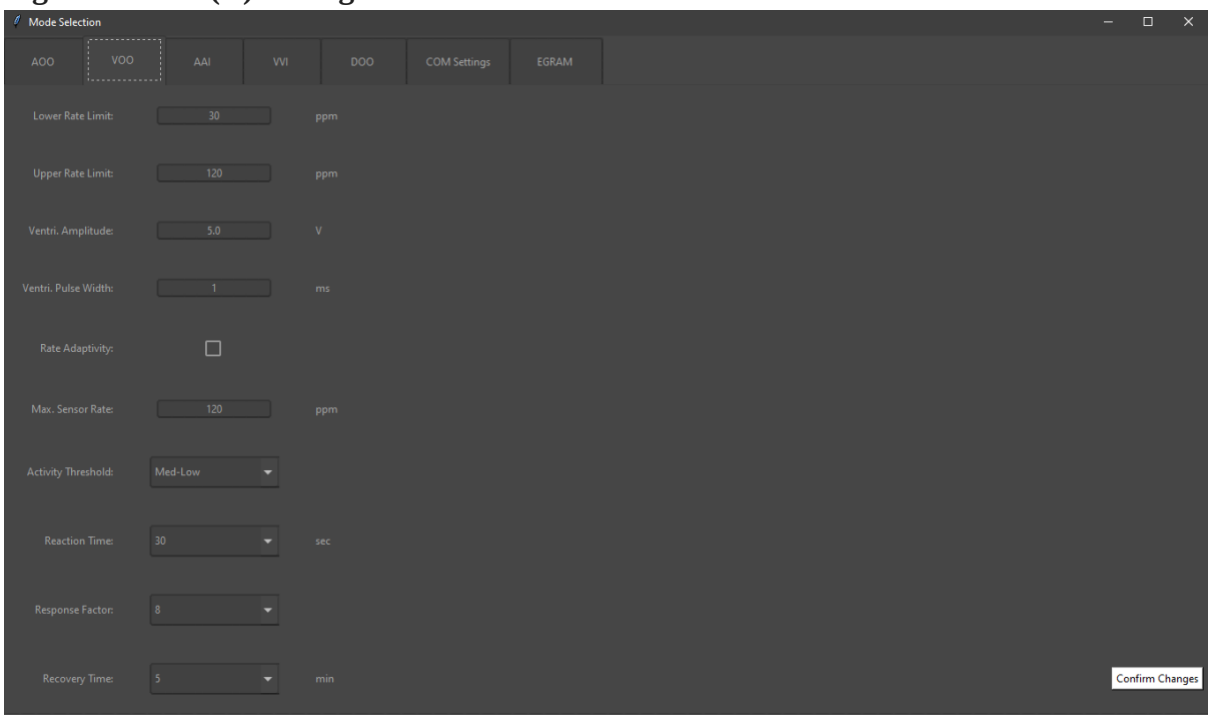

Figure 9: AAI(R) Interface

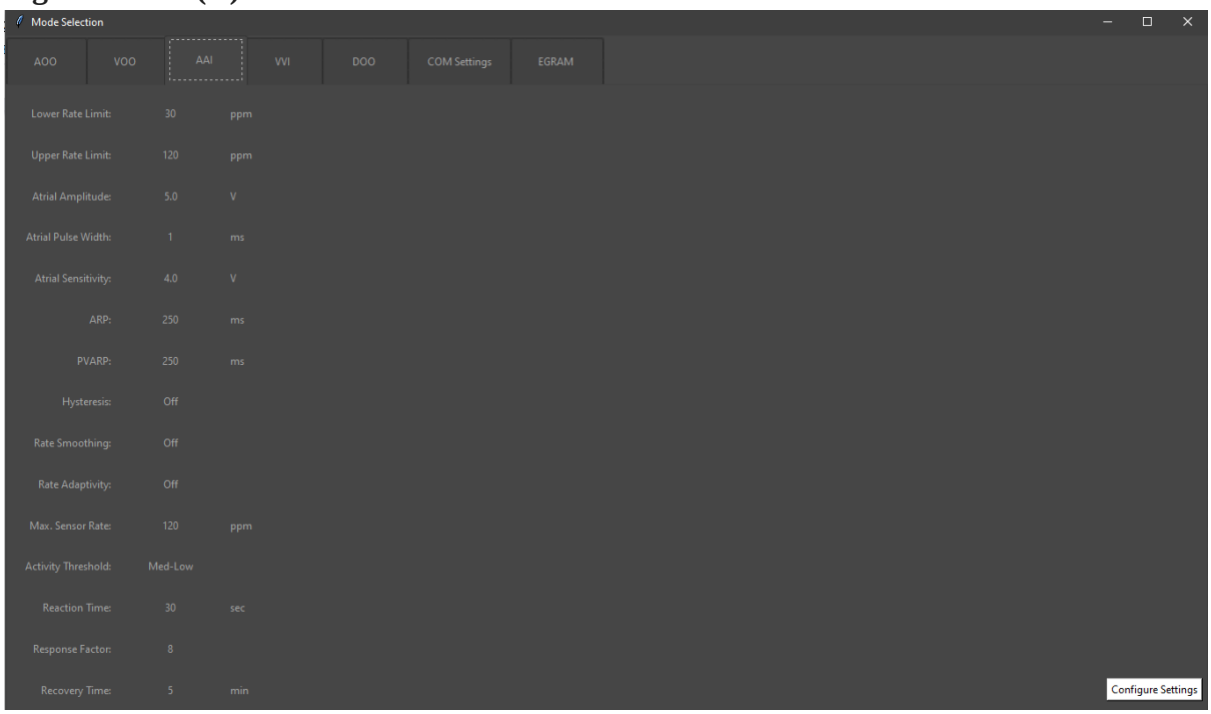

Figure 10: AAI(R) Configuration Menu

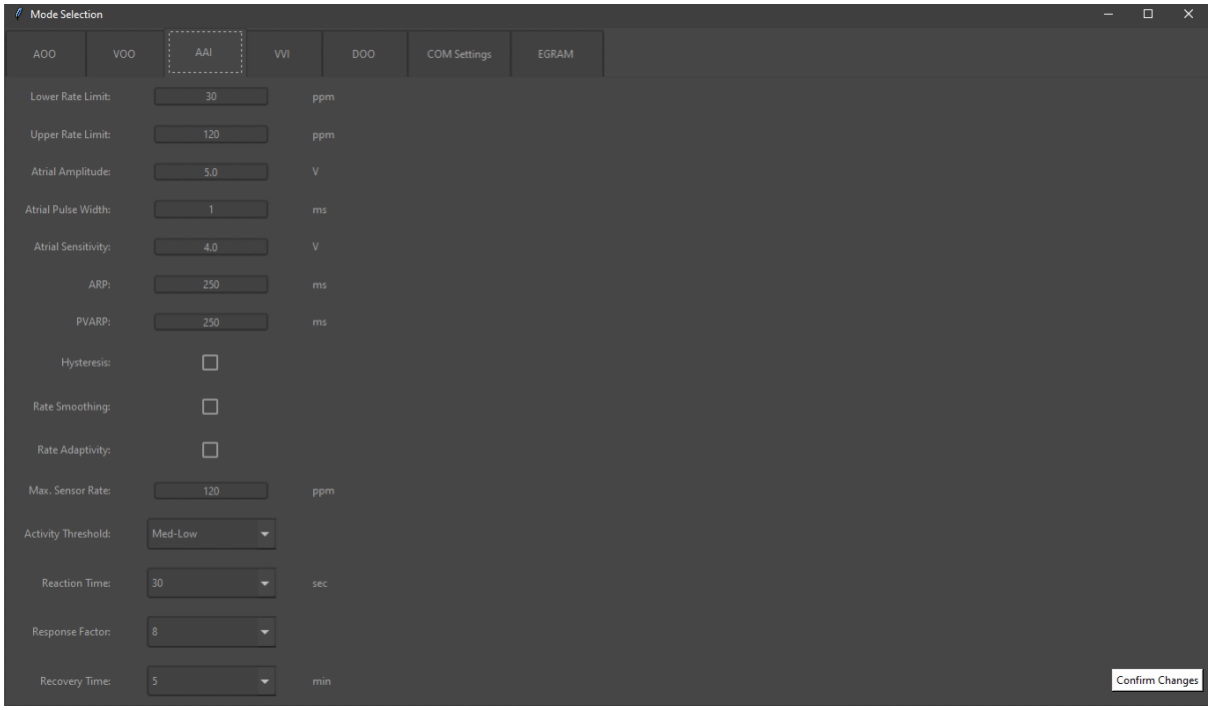

Figure 11: VVI(R) Interface

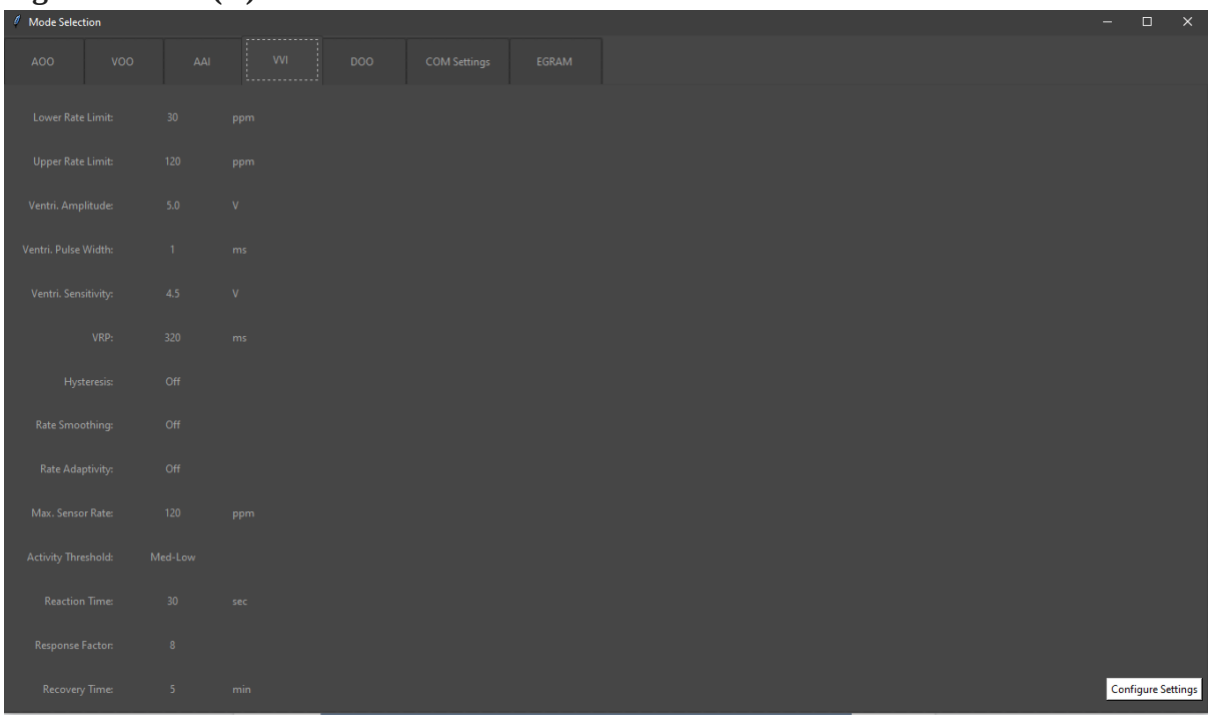

Figure 12: VVI(R) Configuration Menu

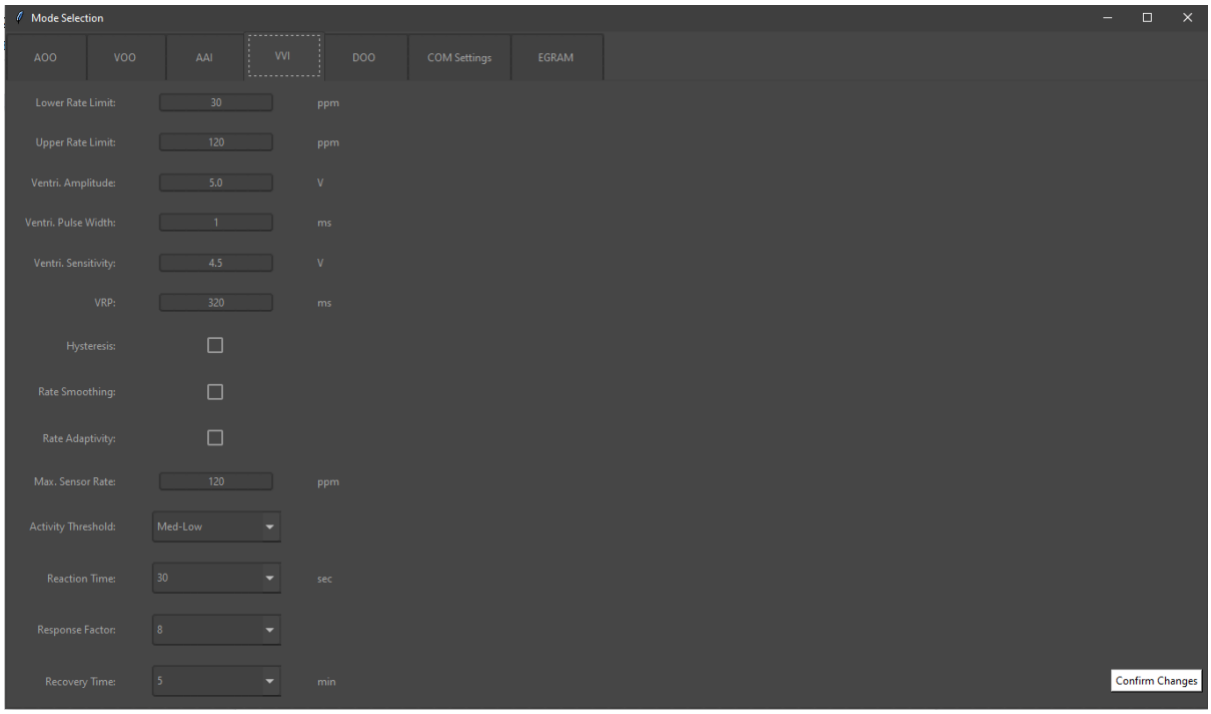

Figure 13: DOO(R) Interface

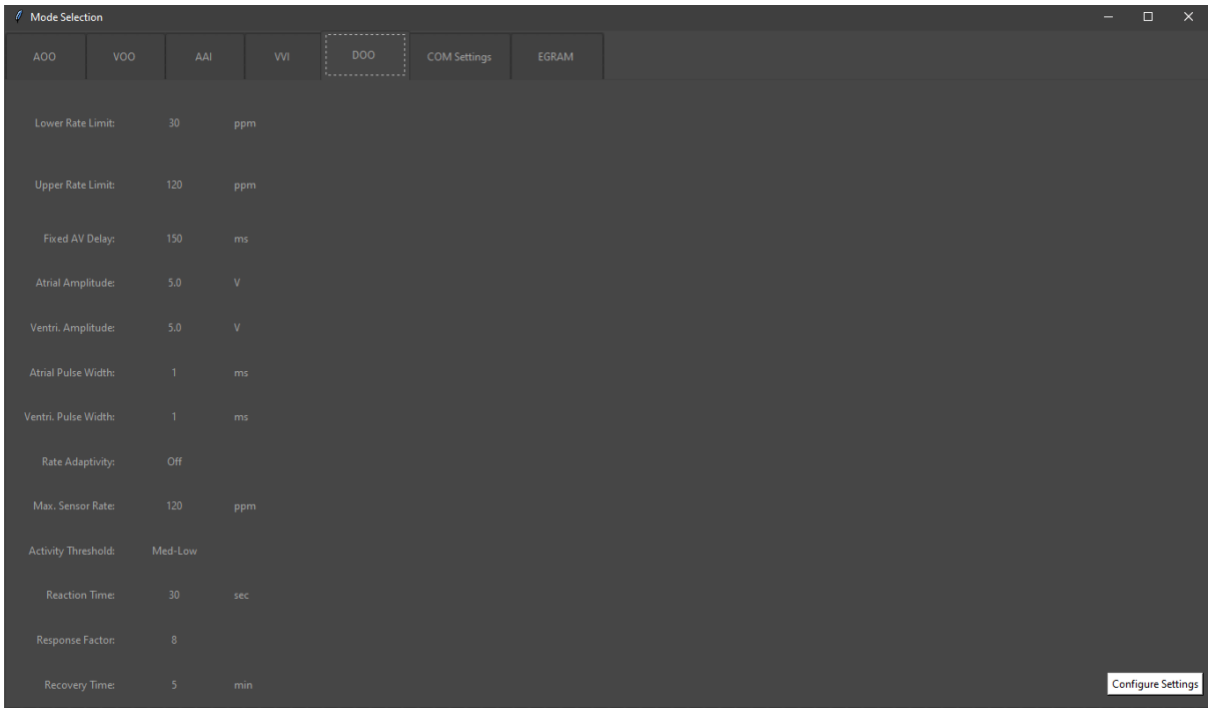

Figure 14: DOO(R) Configuration Menu

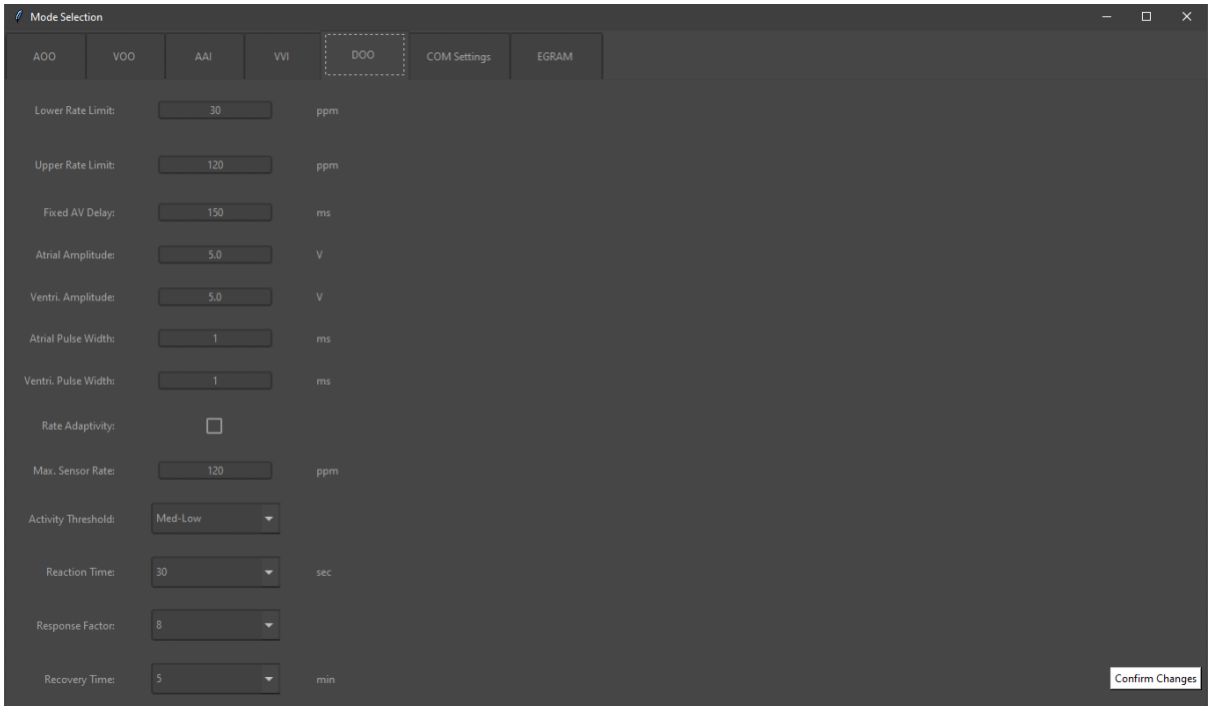

Figure 15: COM Settings Menu

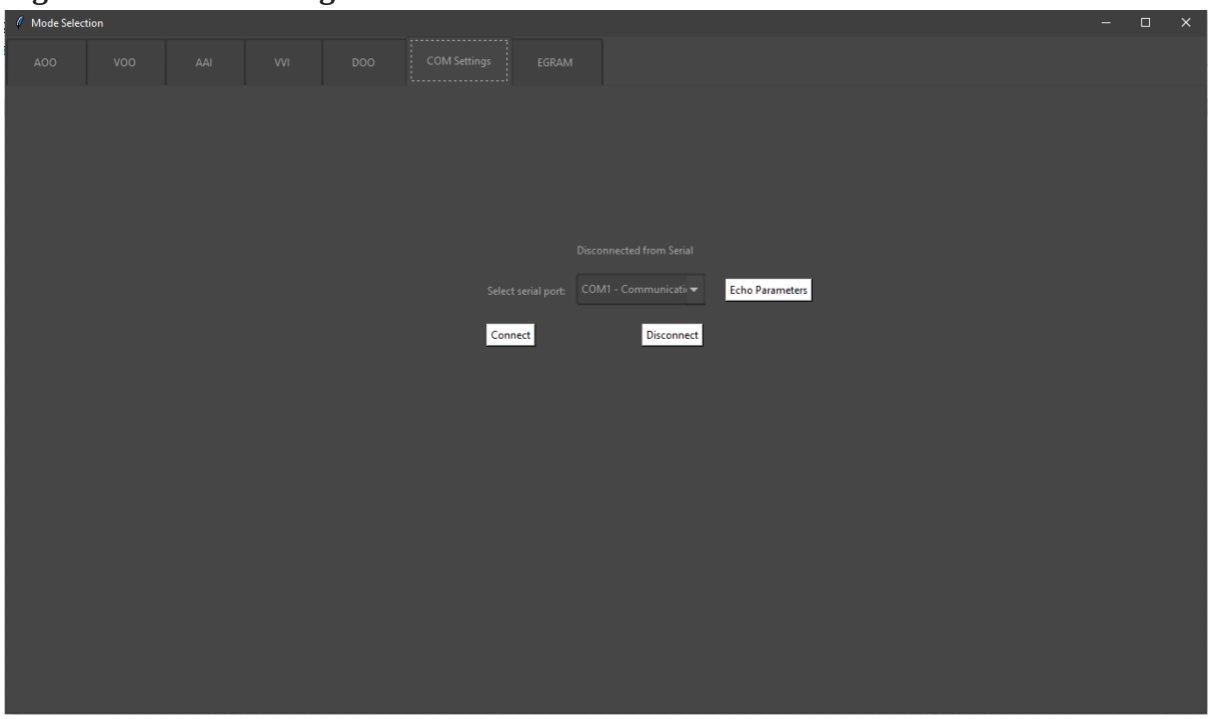

Figure 16: Serial Communication Port Selection

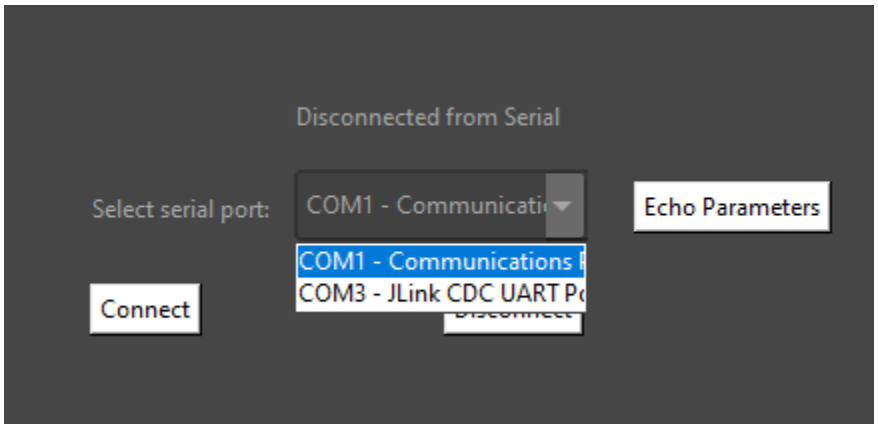

Figure 17: Egram Interface

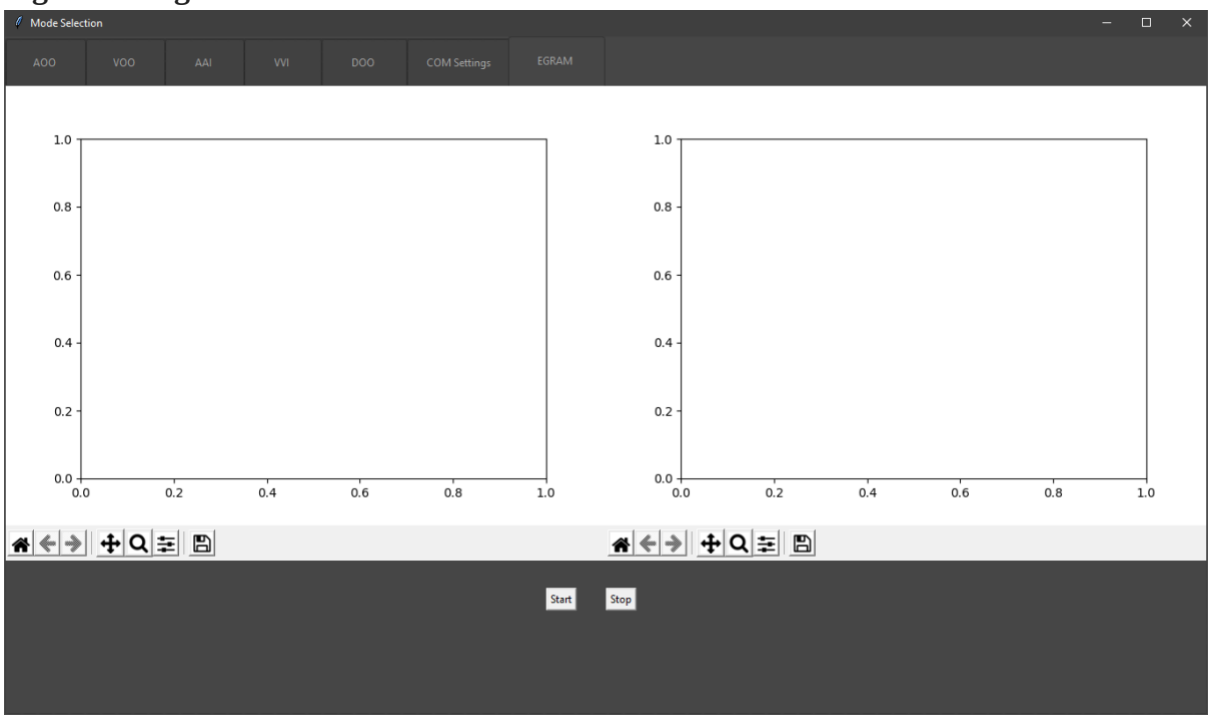

Figure 18: Egram running, no activity