Alpha & Beta Softwares

User and Programmer Manual

Version 1.0

June/2015

University of Reading

School of Systems Engineering

Brain Embodiment Lab

Johnathan M Melo Neto & Vitor de Carvalho Hazin

1. Introduction

This document was written in order to provide detailed information about the two main softwares that are part of the recreated animat from the University of Reading: Alpha and Beta. All the basic information that the developers and users need in order to use and modify these programs are described here. It is important to mention that the current status of the softwares is not definitive, ergo, they will need to be updated during the researchprogress. That's why this manual is needed.

2. Overview of the softwares

The aim of the system is to provide a complete integration between a Multi Electrode Array (MEA) with neuronal cells and a mobile robot. The biological signals of the cells will be acquired by a software named MC_Rack, from the MultiChannel Systems. When MC_Rack detects specific signal patterns on the electrodes, it creates a *.mcd file that contains, among other types of data, a digital bit associated with the channel (electrode) that reached a predefined voltage threshold. A MATLAB script was developed in order to read every *.mcd file that is created, and evaluate, for each of them, which channel was triggered. When the script acquires this information, it creates a *.txt file, that contains the digital bit, the channel ID, and the number of the *.mcd file that was read.

After the generation of the *.txt files with the information of the channels, the Alpha software is the next step of the closed-loop pipeline. Alpha detects the *.txt files, read their content and transmit the channel information to Beta via TCP/IP socket. Alpha always must be in the same computer that the MC_Rack is, since they communicate with each other via files. The TCP/IP socket communication between Alpha and Beta was chosen since it provides a reliable way to transmit and receive data wireless, with no loss in a reasonable time spam.

Beta receives the information from Alpha and, based on a given decoding algorithm, it computes the wheel speeds of the mobile robot. Beta was built to command a real mobile robot and a simulation robot as well. The simulation platform used is V-REP, which has the 3D model of the robot. Currently, only the simulated robot is used. The communication between the Beta and the real mobile robot will be implemented. It is important to mention that Beta should be in a computer near the real robot to avoid communication delays, since the communication between the laptop and the robot is often via Bluetooth.

When the robot reaches the vicinity of an obstacle (e.g a wall), the distance sensors detect the obstacle and trigger an event in Beta. When Beta receives the information that an obstacle has been detected, it sends this message to Alpha via TCP/IP. Alpha, when receives this information, should stimulate the neuronal cells by using another software, MC_Stimulus, that is responsible for stimulate the MEA. The stimulation part in Alpha is will be implemented. The above description is the completed closed-loop of the system. Figure 1 illustrates this process.

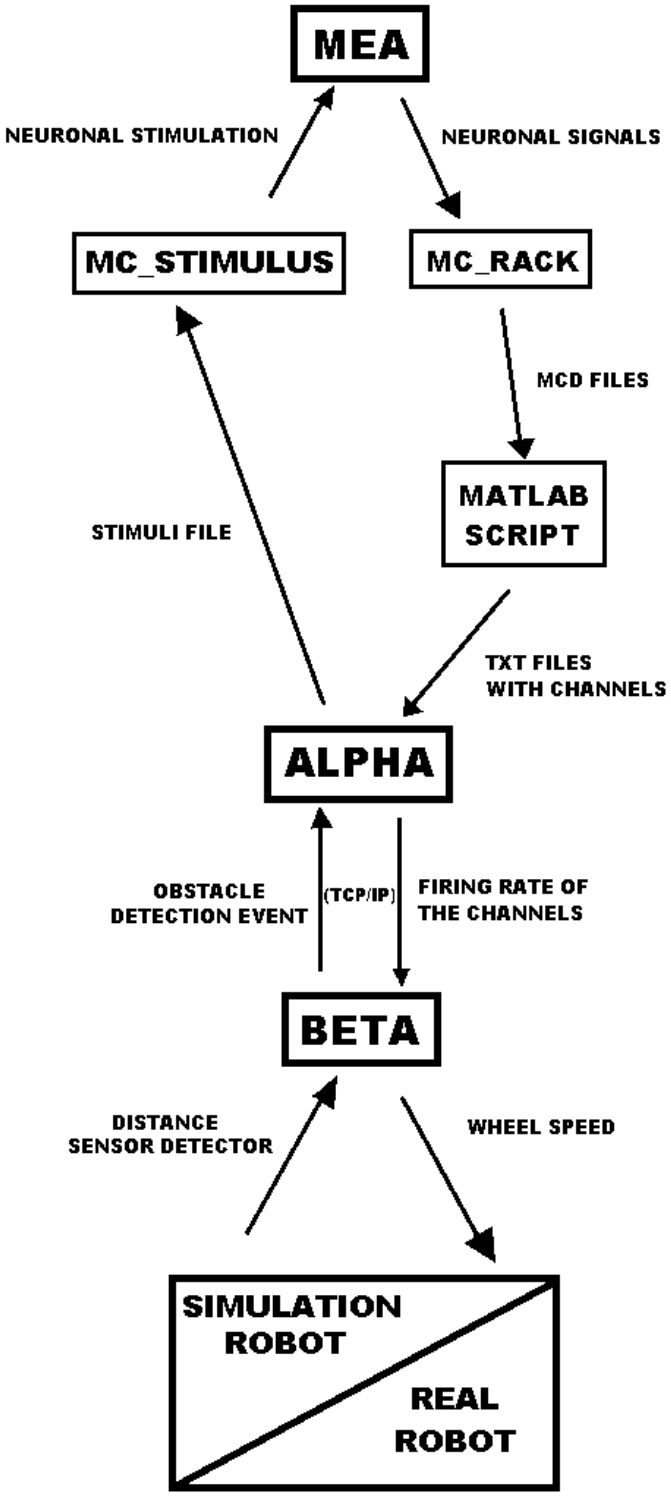

FIgure 1. Diagram of the closed-loop system.

Both applications are based on a multithreaded approach, since threads provide an efficient and fast way to process simultaneously all the data that flow in the closed-loop. The programming language used was C++, because it provides more flexibility to modify whatever it takes, it is easy to use, and the V-REP simulator has an external remote C/C++ API with functions that allow us to control the simulation via software. The Alpha and Beta will be explained in more details, separately, in the next sections.

3. ALPHA

3.1 How Alpha works

First of all, when Alpha is initialized, only the *main* thread is active. The main purposes of the *main* thread are: initialize some variables; connect to Beta via TCP/IP; start the timer event; open all the other threads and wait for their finalization. The Alpha only starts to exercise its function when all threads are opened by the *main* thread. The connection between Alpha and Beta are made by TCP/IP by using the sockets API in Windows, the Winsock version 2.2, that has lots of functions that allow the communication between two different programs. The communication is based on a client/server approach, where Alpha is the client and Beta is the server. The client needs the IP address of the server to establish the connection. Once the IP is provided, Alpha tries to connect with Beta. If the connection is performed successfully, the timer event is started. Once started, this timer is triggered every 1000 milliseconds in order to count how many times the channels related to the left wheel and right wheel were fired, so one can have the firing rate of these channels in Hertz. After the initialization of the timer, all the threads are opened, and the *main* thread waits for their finalization. When all the threads finished, the *main* thread closes all the synchronization objects, the socket, and others. When Alpha finishes, it creates a txt file for offline analysis that can be visualized by the m-file *plotFiringRate* in Matlab. Figure 2 illustrates how the *main* thread works.

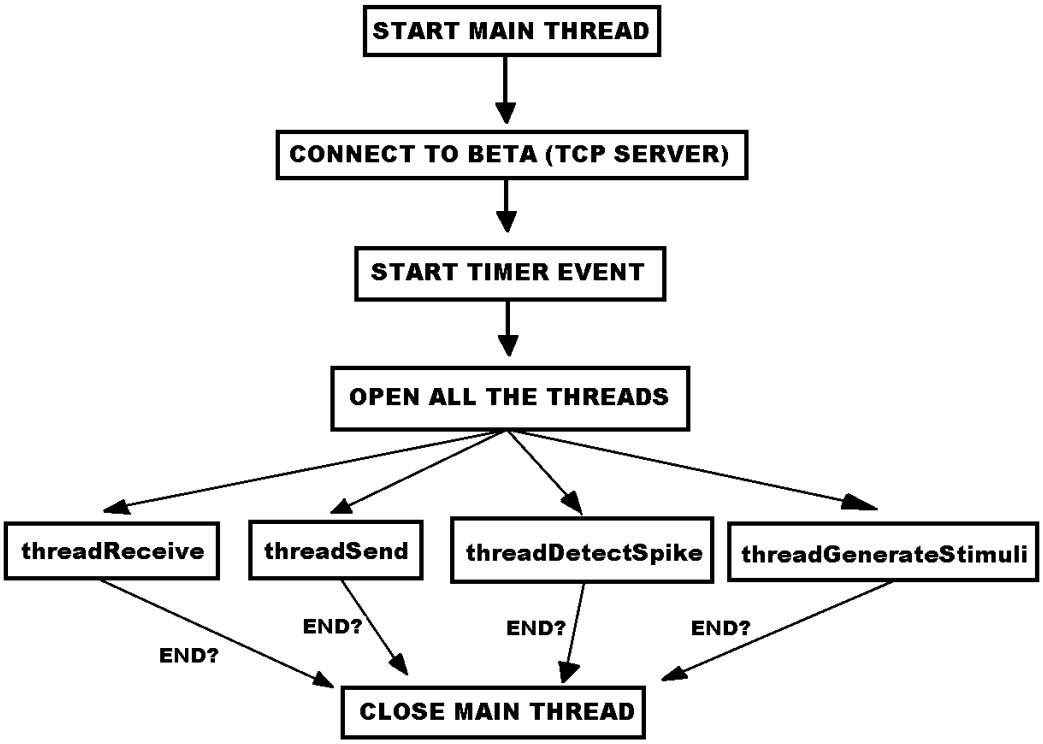

Figure 2. Diagram of ALPHA illustrating how the initialization of the threads works.

After the initialization of the threads, they started to work independently, and the *main* thread gets in a waiting status. The 4 threads that start to work are: *threadGenerateStimuli*, *threadDetectSpike, threadReceive,* end *threadSend.* The *threadDetectSpike* is responsible for the reading of the txt files that the Matlab Script has generated. If the file exists, the thread reads its content in order to decide which channel was fired. If the first character of the txt file is "1", then the counter related to the channel responsible for moving the left wheel, *leftChannel,* is incremented. Similarly, if the first character of the txt file is "2", then the counter related to the channel responsible for moving the right wheel, *rightChannel,* is incremented. These are global variables, so other parts of the program can access their content. The timer, as explained above, is triggered every second. When it triggered, it reads how many times the *leftChannel* and the *rightChannel* were incremented. Then, the timer stores the content of these variables in other variables, the *leftFiringRate* and *rightFiringRate,* respectively. As the timer is fired every second, these variables represent the firing rate (in Hertz) of the channels related to the left wheel and right wheel. They are global variables as well, so the *threadSend* can send them to Beta. The timer also writes in a txt file the values of the *leftFiringRate, rightFiringRate, stimuliLeft,* and *stimuliRight* variables, for Matlab offline analysis purpose. The *threadSend* converts the *leftFiringRate* and *rightFiringRate* into the string type, so it can send them to

Beta as a textual message. To differentiate between the left firing rate and the right firing rate, the message begins with "L", if the number refers to the left firing rate, or it begins with "R", if the number refers to the right firing rate; therefore Beta can read the message and easily identify to which channel the received number refers to. For example, if the *leftFiringRate* is 25, the sent message will be "L025", if the *rightFiringRate* is 8, the sent message will be "R008".

The threadReceive is responsible for receiving any message that Beta sent. The Beta can send the messages "0000", "3000", and "4000". If the thread receives the message "3000", it means that the mobile robot has found an obstacle due to the left distance sensor, so the *stimuliLeft* counter is incremented. Similarly, if the thread receives the message "4000", it means that the mobile robot found an obstacle due to the right distance sensor, so the *stimuliRight* counter is incremented. These counters are global variables, therefore the *threadGenerateStimuli* can access them and stimulate the channels related to the left wheel or right wheel. The *threadGenerateStimuli*, as mentioned, detects if the *stimuliLeft* and the *stimuliRight* were incremented, if one of them was incremented (greater than zero), the thread is supposed to stimulate the channel related to the left wheel or the right wheel (depending of which counter was incremented). This stimulation part was not implemented yet, so just a message is written in the prompt so the user can check which variable was incremented, but, in the future, the thread should generate stimuli files in order to stimulate the MEA. The Figure 3 shows the data flow of Alpha, as explained above.

If the *threadGenerateStimuli* receives the message "0000", it means that either the ENTER button or the ESC button was pressed in Beta. This is important to synchronize the *threadDetectSpike* and the *threadSend*. For example, if the ENTER button is pressed in Beta, it means that the user started the robot, so the two above threads can work normally, that is, the *threadDetectSpike* can read the txt files, and the *threadSend* can send the firing rates of the channels to Beta. If the ESC button is pressed in Beta, it means that the user stopped the robot, so both threads can stop working, that is, the *threadDetectSpike* can stop reading the txt files, and the *threadSend* can stop sending the firing rates of the channels, since the robot is stopped, so these informations are no longer useful. The global variable used to synchronize this functioning is *bStart*, that can be true if the ENTER button is pressed in Beta, or it can be false if the ESC button is pressed in Beta. The *threadGenerateStimuli* is not affected by the *bStart*. The Figure 4 illustrates this control data flow from Beta to Alpha.

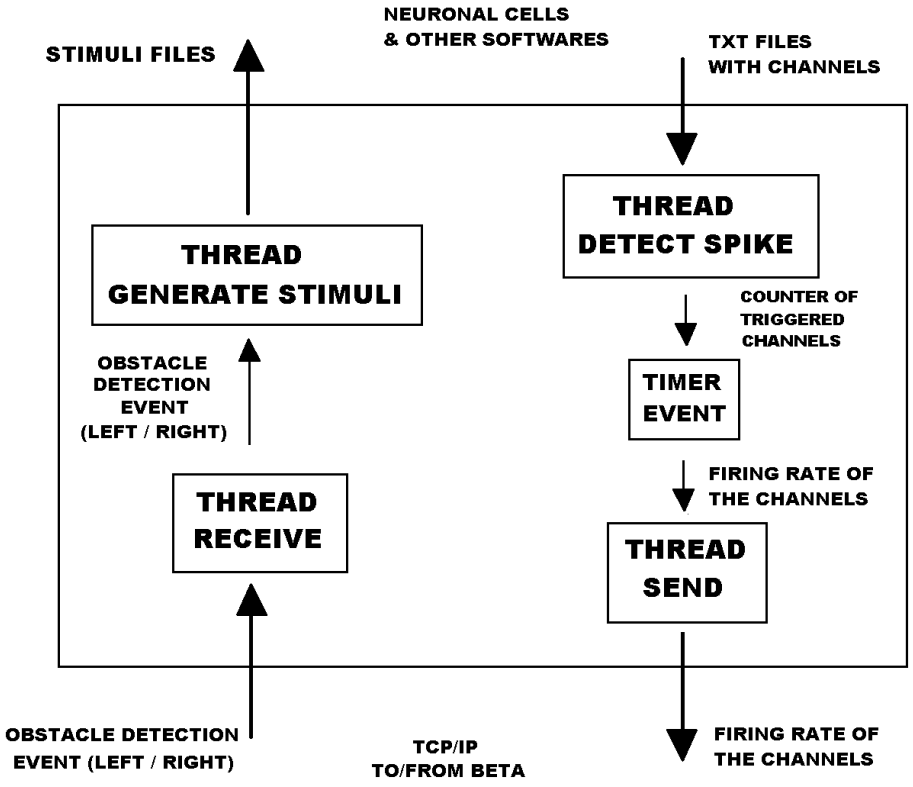

Figure 3. Diagram of ALPHA illustrating its main functionality.

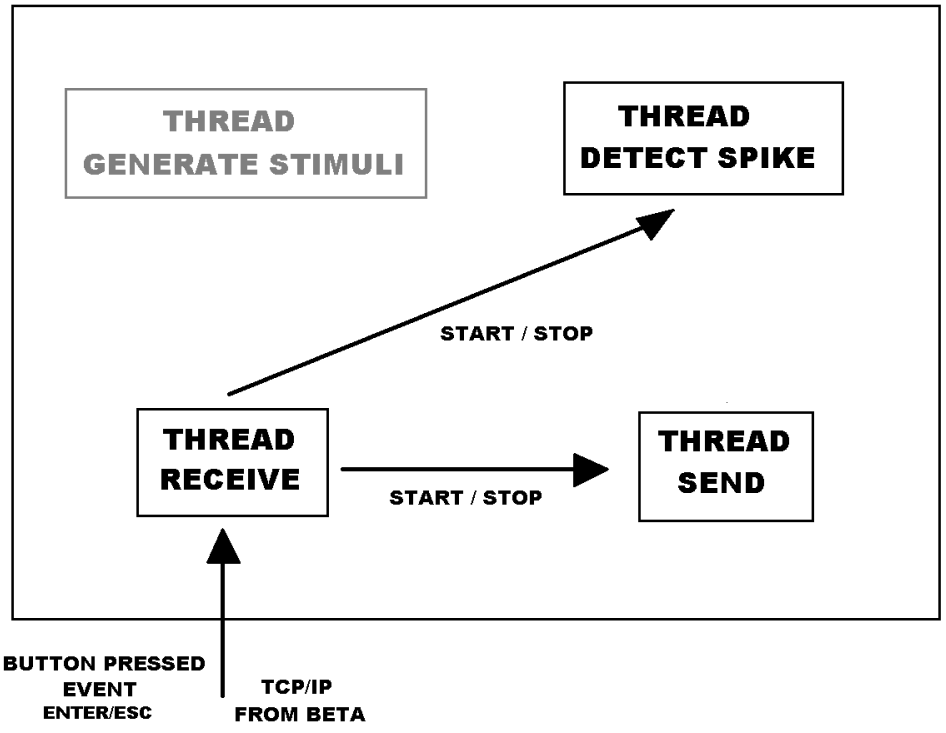

Figure 4. Diagram of ALPHA illustrating how the user can pause/unpause the data flow.

3.2 Description of the threads

This section provides a detailed description of each thread:

void main() – The *main* thread is the first to run. It enables the event handler *ConsoleHandler* to detect if the user closed the console application, it is responsible for the initialization of the synchronization objects (mutexes), for the initial configurations, for the creation of the txt file for offline analysis purpose, for the initialization of the timer event, for the connection between the ALPHA (TCP Client) and BETA (TCP Server), and for the initialization of the other threads. When the main thread opens the other threads, it waits for their finalization indefinitely. It is important to mention that, when Alpha is trying to establish a connection with Beta, it performs three attempts, only. If the connection is not made, the program aborts itself. When all the threads finished, the *main* thread closes all the synchronization objects, the socket, the txt file, and so on.

DWORD WINAPI threadReceive(LPVOID) – This thread has a single function: it receives messages from the BETA application. The three types of messages that can be received are: 1) Obstacle detected due to the left sensor of the robot ("3000"); 2) Obstacle detected due to the right sensor of the robot ("4000"); and 3) The start/stop button in BETA was pressed ("0000"). The two first messages are useful to know the exact time that a stimulation in the neuronal cells must be performed, and how (in which channels).

DWORD WINAPI threadSend(LPVOID) – This thread has a single function: it sends messages to the BETA application. The message types that are sent to BETA are the firing rate of the channels related to the left wheel speed, and the firing rate of the channels related to the right wheel speed of the robot. The structure of the message is “Lxxx” or “Rxxx”; the “L” refers to the left wheel, the “R” refers to the right wheel, and the “xxx” is the value of the firing rate (firings per second).

DWORD WINAPI threadDetectSpike(void) – This thread is responsible for reading txt files generated by a Matlab Script in order to identify which channels are responsible for a given spike. The files can be differentiated by its first character. If the first character of the txt file is "1", then the counter related to the channel responsible for moving the left wheel, *leftChannel,* is incremented. Similarly, if the first character of the txt file is "2", then the counter

related to the channel responsible for moving the right wheel, *rightChannel,* is incremented. These counters are global, so one can count how many firings they are generated per second.

DWORD WINAPI threadGenerateStimuli(void) – This thread is responsible for the generation of the stimuli files in order to induce a stimulus in specific channels of the MEA based on the messages that the *threadReceive* receives from BETA. This thread is not completely implemented, since it is not generating the stimuli files yet. If the *stimuliLeft* is positive, then a stimuli in the channels related to the left wheel should be performed. Similarly, if the *stimuliRight* is positive, then a stimuli in the channels related to the right wheel should be performed.

3.3 Description of the functions

This section provides a detailed description of each function:

void OpenThreads (int) – This function is placed in the *main* thread, and it creates all the other threads and waits for their finalization.

void configurationScreen(void) – This function was implemented for future configurations that the ALPHA application must need.

void TimerEvent::StartTimer() – When enabled, this function starts a timer with a predefined interval.

void TimerEvent::OnTimedEvent(Object^ source, ElapsedEventArgs^ e) – This function is a timer that will be triggered every second to check how many times the variables related to the left and right wheels were fired, so one can have the firing rates of these channels. It also writes on the txt file the necessary information for an offline analysis.

4. BETA

4.1 How Beta works

First of all, when Beta is initialized, only the *main* thread is active. The main purposes of the *main* thread are: initialize some variables; configure initial parameters; connect to V-REP simulation; connect to Alpha via TCP/IP; open all the other threads and wait for their finalization. The Beta only starts to exercise its function when all threads are opened by the *main* thread. As Beta is the TCP Server, it waits indefinitely by the contact of Alpha (TCP Client). When all the threads finished, the *main* thread closes all the synchronization objects, the sockets, and others. It is important to notice that the Beta can be only opened inside the V-REP simulation. When the Play button is pressed in the simulation, Beta starts automatically. Therefore, the Beta exe file must be inside the V-REP folder. Figure 5 illustrates how the *main* thread works.

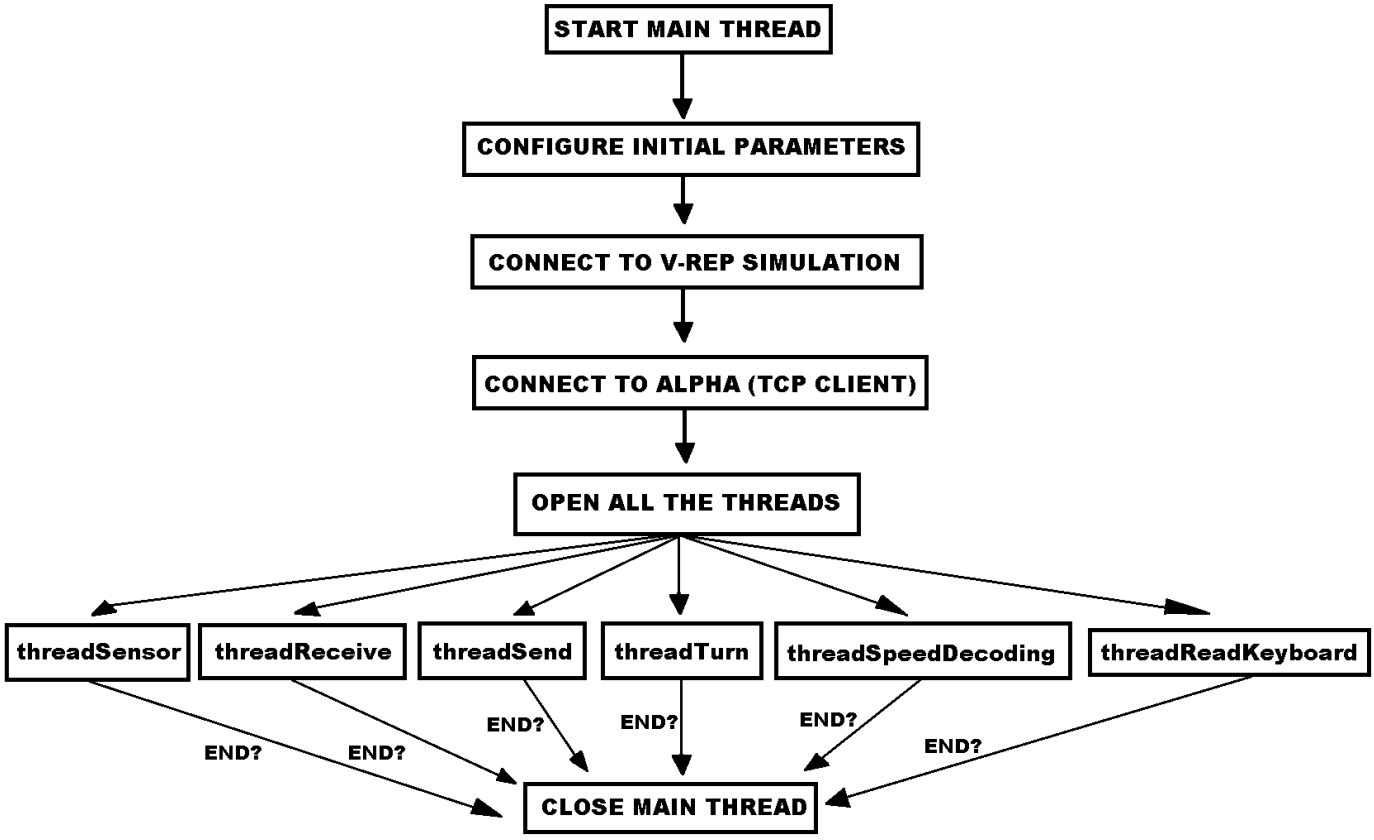

Figure 5. Diagram of BETA illustrating how the initialization of the threads works.

After the initialization of the threads, they started to work independently, and the *main* thread gets in a waiting status. The 6 threads that start to work are: *threadReceive, threadSpeedDecoding, threadTurn, threadSensor, threadReadKeyboard* and *threadSend.* The *threadReceive* is responsible for receiving messages from Alpha. As aforementioned, the types of messages that Beta can receive are: "L _ _ _" and "R _ _ _", where the first position of

each message refers to the channel related to the left wheel and to the right wheel, respectively, and the other three positions are filled with the firing rate of the channel (in Hertz). Therefore, the thread receives the message, stores its content in a variable named *buf*, and it reads its first position (*buf[0]*) in order to differentiate between the left firing rate and the right firing rate. After the differentiation, the thread stores the firing rate value in specific global variables: *leftFiringRate* or *rightFiringRate*.

These variables are used by the *threadSpeedDecoding* to convert the firing rates into values of the left wheel speed and the right wheel speed. This thread is just a switch-case that has many decoding methods inside. The speed decoding method can be chosen during the initial configurations of Beta, when the variable *decodingMethod* is modified accordingly to the chosen method. Currently there are two available methods: the discrete decoding (*discreteDecoding*) and the winner-takes-all (*WTADecoding*). These methods will be explained later. The important points now are: this thread is a switch and allows the insertion of other decoding methods; the user can select which one he wants during the initial configurations of Beta; the decoding method generates two outputs (the speed of the left wheel and the right wheel), and it stores these values in the global variables *leftWheel* and *rightWheel*, respectively. These variables will be used by *threadTurn.*

The *threadTurn* is responsible for sending the wheels' speed to the mobile robot simulation. It uses the *leftWheel* and *rightWheel* variables in order to control the robot. The thread uses the V-REP remote API functions that can control the simulation. The details of these functions can be found at <http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctions.htm>. The *startRobot* variable can control if the user pressed the ENTER or ESC button, in order to control the robot. If the user presses ENTER, the robot can move. However, if the user presses the ESC button, the robot stops, and if the user presses the SPACEBAR, the robot moves backwards for 2 seconds, for safety purpose.

The *threadSensor* is responsible for reading the robot distance sensors. The current simulated robot (K-Junior) has 5 sensors in different positions: left, left-center, center, right-center, and right. We use the remote V-REP API function *simxReadProximitySensor* to perform the reading of each sensor. The obstacle detection is useful to know if the obstacle was detected on the right or on the left side of the robot. This information is important since we can stimulate the channels responsible for the left wheel and the channels responsible for the right wheel. If the sensors detect an obstacle on the left side, the global variable *doStimuliLeft* is incremented. Similarly, if the sensors detect an obstacle on the right side, the global variable *doStimuliRight* is incremented. This information will be sent to Alpha in order to perform the proper stimuli. There are two types of coding that can be implemented in this

thread: the binary coding and the proportional coding. The binary coding is a method that waits a fixed period of time between two stimuli request. The proportional coding is a method that waits a variable period of time between two stimuli request; the closer the robot is from the obstacle, the lesser is the time period to wait until the next sensors reading, and vice-versa. This waiting period between the sensors reading is important to avoid excess of stimuli requests on the channels of the MEA. It is unsafe stimulate the neuronal cells without waiting a minimum period of time, generally 500 milliseconds. The details of implementation of the sensors readings, of the binary coding method, and of the proportional coding method will be explained later.

The *threadSend* is responsible for sending Alpha the messages to stimulate the channels related to the left wheel or to the right wheel. The thread reads the content of the global variables *doStimuliLeft* and *doStimuliRight*. The message "3000" is sent if the *doStimuliLeft* is greater than zero, the message "4000" is sent if the *doStimuliRight* is greater than zero. Alpha receives these messages and differentiates them in order to stimulate the proper channels. The Figure 6 illustrates the data flow from Alpha to Beta, from Beta to the robot, and vice-versa.

The *threadReadKeyboad* is responsible for reading the keyboard of Beta. This is useful because the user can control the robot movements utilizing some buttons. If the user presses the ENTER, the boolean variable *startRobot* is true, and the robot starts reading the sensors and starts receiving the wheel speeds to move. If the user presses the ESC, the boolean variable *startRobot* is false, and the robot stops reading the sensors and it stops receiving the wheel speeds, so it stops. If the user presses the SPACEBAR, the boolean variable *spacePressed* is true, and then the robot goes backward during 2 seconds. Moreover, when the user presses the ENTER or ESC, this event must be sent to Alpha in order to stop some of its threads (Figure 4). The *threadReadKeyboard,* when detects that the ENTER button or the ESC button were pressed, it changes to true the boolean variable *buttonPressed,* so the *threadSend* sends the message type "0000" to Alpha in order to communicate if the user started or stopped the robot. The Figure 7 illustrates this procedure.

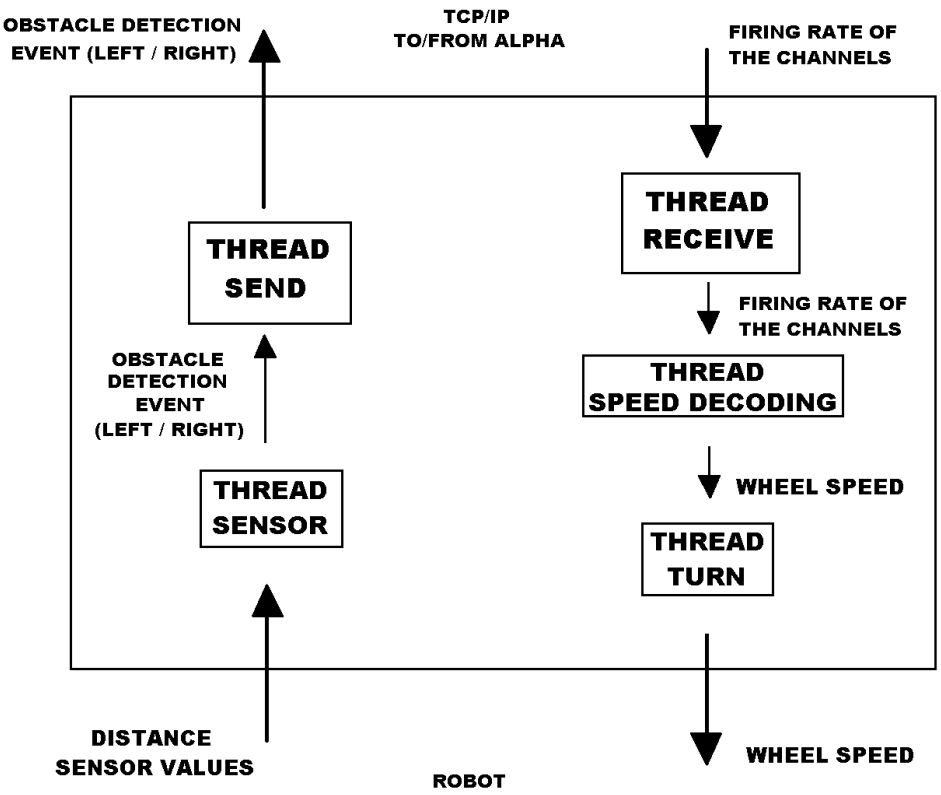

Figure 6. Diagram of BETA illustrating its main functionality.

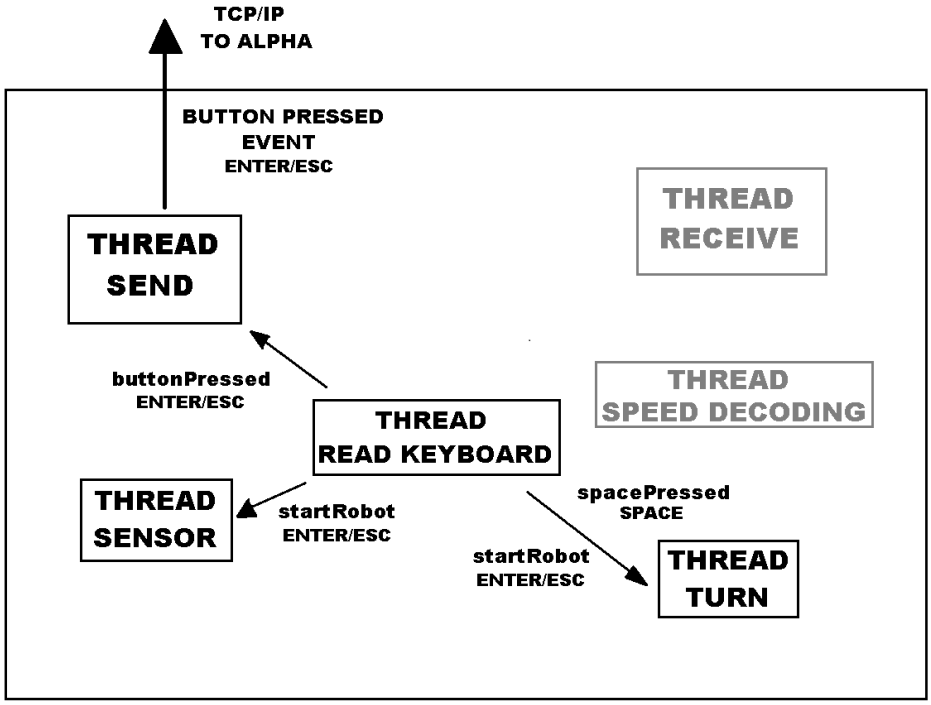

Figure 7. Diagram of BETA illustrating how the user can pause/unpause the data flow.

4.2 Description of the threads

This section provides a detailed description of each thread:

int main(int argc, char *argv[]) – The *main* thread is the first to run. It enables the event handler *ConsoleHandler* to detect if the user closed the console application, it is responsible for the initialization of the synchronization objects (mutexes), for the initial configurations, for the connection between the Beta and the V-REP platform,for the connection between the Beta (TCP Server) and Alpha (TCP Client), and for the initialization of the other threads. When the *main* thread opens the other threads, it waits for their finalization indefinitely. When all the threads finished, the *main* thread closes all the synchronization objects, the sockets, the simulation, and others.

DWORD WINAPI threadReceive(LPVOID) – This thread has a single function: it receives messages from the Alpha application. The message types that the Beta can receive are “Lxxx” and “Rxxx”, as aforementioned. Based on these messages, the firing rates corresponding to each wheel are stored in specific global variables to be manipulated by the *threadSpeedDecoding*.

DWORD WINAPI threadSend(LPVOID) – This thread has a single function: it sends messages to the Alpha application. As mentioned before, the three types of messages that can be sent are: 1) Obstacle detected due to the left sensor of the robot ("3000") when the global counter *doStimuliLeft* is greater than zero; 2) Obstacle detected due to the right sensor of the robot ("4000") when the global counter *doStimuliRight* is greater than zero; and 3) The start/stop button in Beta was pressed ("0000") when the global boolean *buttonPressed* is true.

DWORD WINAPI threadTurn(LPVOID) – This thread utilizes the wheel speeds that the *threadSpeedDecoding* computed in order to send these commands to the V-REP robot. First of all, the thread gets the handles of the robot's motors (left and right). These handles will be used to send the speeds to their respective wheels by the remote function *simxSetJointTargetVelocity*. The global boolean variable *startRobot* is used to identify when the robot should move and when it should stop. The global boolean variable *spacePressed* is used to identify when the robot should move backwards during 2 seconds.

DWORD WINAPI threadSensor(LPVOID) – This thread acquires the distance sensors values of the robot, identifies which sensor was responsible for the detection of the obstacle, and increments the global counters *doStimuliRight* or *doStimuliLeft,* depending on which sensor detected the obstacle. Initially, the thread gets the handles of the 5 sensors using the remote function *simxGetObjectHandle.* As aforementioned, the 5 sensors are placed in the left,

left-centre, centre, right-centre, and right of the robot. For each sensor, the thread uses the remote function *simxReadProximitySensor* in order to identify if the sensor detected something and, if detected, it reads the distance between the sensor and the obstacle. The vector *sensorOn[i]* is used to identify which sensor detected something and which sensor did not. If the sensor detected some obstacle, the thread reads the distance between the sensor and the obstacle in order to calculate the waiting period between two consecutive stimuli (if proportional coding was chosen). The boolean variable *binaryCoding* is used to identify if this waiting period will be fixed (binary coding, boolean is true) or variable (proportional coding, boolean is false). After these calculations, the thread need to identify on which side of the robot the obstacle was found. The vector *sensorOn[i]* is used in order to achieve this aim, since this vector has the information of which sensors detected an obstacle. By using this strategy, the thread is able to increment the counters *doStimiliLeft* and the *doStimuliRight* properly.

DWORD WINAPI threadReadKeyboard(LPVOID) – This thread performs a single task: it constantly reads the keyboard. There are three buttons that the user can press in order to control the robot. The ENTER starts the robot, the ESC stops it, and the SPACEBAR moves the robot backwards, in case of a dangerous position that the robot is placed (e.g. near an edge).

DWORD WINAPI threadSpeedDecoding() – Thread that processes the firing rates and translates them into motor commands (wheel speeds). There are lots of decoding methods that can be implemented in order to compute the speed of each wheel of the robot based onthe firing rates of the channels. So this thread is simply a switch-case that allows other methods to be inserted/implemented.

4.3 Description of the functions

This section provides a detailed description of each function:

void OpenThreads (int ClientSock, int clientID) – This function is placed in the *main* thread and it creates all the other threads.

void configurationScreen(void) – This function creates a configuration screen in the console so the user can change some parameters: maximum robot speed, speed decoding method, and external sensory input patterns.

void configurationDecodingMethod(void) – If the user intends to change the speed decoding method, this allows the user to change which decoding method will be used when the application runs.

void configurationMaximumSpeedRobot(void) – If the user intends to change the maximum speed of the robot, this function allows the user to change this parameter.

void configurationSensoryPatterns(void) – If the user intends to change the external sensory input patterns, this function allows the user to change which pattern will be used when the application runs.

void adjustParameters(void) – If the user intends to change the coefficients of the winner-takes-all algorithm, this function allows the user to change them.

int discreteDecoding() – This is one of the two speed decoding methods that was already implemented. The discrete decoding method sends discrete commands to the robot (that is, "turn left" or "turn right") based on the detection of the left arrow and the right arrow (in Alpha). This method is used mainly for testing purpose and should be removed soon.

int WTADecoding() – This is one of the two speed decoding methods that was already implemented. The Winner-Takes-All method is based on the instantaneous firing rate of the recording sites. These recording sites are divided into two groups, respectively used for controlling the left and right wheel. The algorithm is based on a research paper published in 2007 by Martinoia.

int connectSimulation(int clientID, int portNb) – This function is used to perform a connection between the Beta and the V-REP platform. This connection is based on a client/server approach. Remote API functions from Coppelia Robotics were used.

SOCKET socketInitializationALPHA(SOCKET ServSock, SOCKADDR_IN ServerAddress) – This function initializes some Socket configurations in order to perform a connection between Beta and Alpha.

int min2(int a, int b) – This is a function that simply returns the smaller number between two.

int min5(int a, int b, int c, int d, int e) – This is a function that simply returns the smaller number between five.

5. How to use the entire system

To connect the system, the following steps has to be done:

5.1 Matlab

1-Open script_canais.m and change the path of the folder that you want to place the txt files.

2-Change four paths corresponding to the mcd files that will be generated by the neurons.

3-Change the matlab folder to C:\Program Files (x86)\Multi Channel Systems\MC_Rack\MCStreamSupport\Matlab\meatools\mcintfac\@datastrm and run datastrm.m program.

4-Run script_canais.m and click in Add to path. Write the channel which is connected in BIT 1 and the channel of BIT 2 (see in 5.4.5 how to change the channels).

5-If you want to stop the program, just click OK on the window that showed up.

5.2 Beta

1-Go to Google Drive\Pesquisa\Alpha + Beta\Projects\beta\Debug and copy beta.exe to C:\Program Files (x86)\V-REP3\V-REP_PRO_EDU.

2-Go to Google Drive\Pesquisa\Robot Simulation\K-Junior V-REP and copy the latest version of the V-REP simulation to C:\Program Files (x86)\V-REP3\V-REP_PRO_EDU\tutorials\K-Junior V-REP (this folder you have to create in “tutorials”).

3-Open the V-REP simulation in the folder above, and run the simulation.

4-Beta program will appear, configure your desired options using the proper buttons.

5-Press Enter and wait for the connection with Alpha.

5.3 Alpha

1-Open Alpha project in visual studio.

2-Click in alpha (enhanced in figure 8) with the right button of the mouse and select properties.

3-A new window will open, select linker, then general, and in Enable Incremental Linking, change to No.

4-Change the path in threadDetectSpike corresponding to the folder you put on 5.1.1 to read the txt files.

5-After compiling, open alpha.exe and click enter. The program is waiting to enter the IP address of Beta PC.

5- To figure out the IP address of Beta PC, run prompt of command in Beta PC and write ipconfig. See the number in ipv4 of wireless connection and insert in alpha program. For example: “10.30.93.54”.

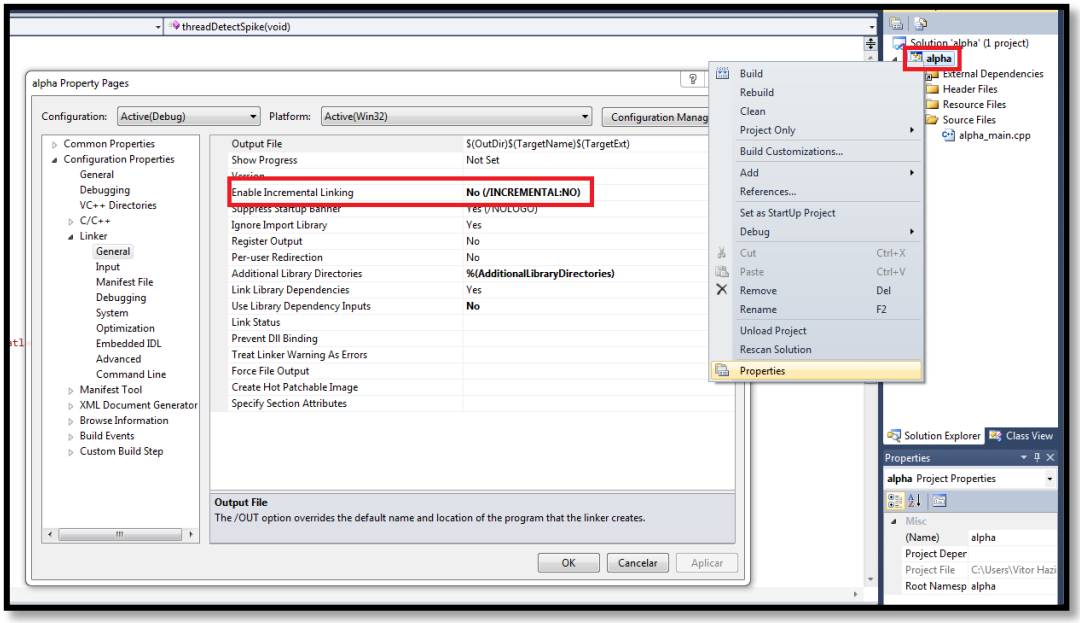

Figure 8. Change of incremental linking in Alpha program.

5.4 Rack

1-Open RealTimeFeedback-07-05-2015 and click in Recorder on the tree window.

2-4 tabs are showing (Rack, Channels, Recorder and Window), click in Recorder.

3-Make sure that Create New File on Trigger is selected.

4-Change the path (the same path you put in 5.1.2) and put the name of datoca.

5-In the tree window, select Real Time Feedback, in one tab shows all the channels and Bits, which you can change.

6-Click on record, the red circle on top and play.

Now, the whole system is working.

When you stop the system, a txt file “firing_rates” is generated in alpha.exe folder with the information about the simulation and responses for Matlab offline analysis. To read this file, open and run the m-file plotFiringRate in Google Drive\Pesquisa\Programa Matlab\Atual (the txt file need to be at the same folder).

5.5 Restart

1-Press stop in Rack, close Beta, Alpha and click OK in matlab program.

2-Write clear all in matlab (so you can erase all the mcd files).

3-Erase all the mcd files and txt files (or change the folder).

4-Do 5.4.4 (every time you restart the system you have to change the name again, sometimes the name become datocaX, which this X is the number of the last mcd file generated).

5-Start the process again.

6. Conclusion

This document was written as a manual for the future programmers that will need to modify the software, and for the users that will need to know how to use the Alpha and Beta in order to perform the necessary experiments using this technology. The document emphasized the data flow of the program between the threads, how each thread works and how they interact with each other. For more details, one must check the C++ code, where there are more comments and details of the implementation.

7. Appendix: considerations about the multi-threading approach

As one can notice, the Alpha and Beta softwares are implemented based on the multi-threading approach. This paradigm is extremely useful since each thread can run independently and at the same time of others. However, this strategy generates some synchronization problems. Imagine that a given variable is used by thread A and thread B. As they work simutaneously, they can write and read on this variable in the same time, which can lead to problems of data inconsistency and data loss. The ideal solution for this problem is the utilization of synchronization objects that allows only one thread per time to manipulate the shared variable. These objects are known as

mutexes, and they are responsible for lock and unlock the usage of a variable or function. The Alpha and Beta utilize the mutexes to synchronize the usage of variables, e.g *doStimuliRight* and *rightWheel* and the usage of some remote API V-REP functions, e.g *simxReadProximitySensor* and *simxSetJointTargetVelocity.* The mutexes are created in the *main* threds of Alpha and Beta, and for each mutex a handle is generated. These handles are the objects that can lock and unlock the variables usage. The function that locks the mutex is *WaitForSingleObject* and the function that unlock the mutex is *ReleaseMutex.* It is important to mentioned that all the mutexes handles, at the end of the application, must be closed.