Universitat Politecnica de Catalunya
Escola Tecnica Superior D'Enginyeria de Telecomunicacio
Departament D'Enginyeria Electronica

MASTER THESIS

of

Adam Sokolnicki

Supervised by PhD. Jordi Madrenas & M.Sc. Giovanny Sanchez Rivera

Graphical representation of data for a multiprocessor platform emulating spiking neural networks

Politechnika Łódzka

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona, July 2010

Acknowledgments

First of all I am greatly thankful to my parents Jan and Malgorzata Sokolniccy in whom I could always find support and help.
Dziekuje Wam bardzo.

Secondly to prof. Jordi Madrenas who approved my arrival and work in the Universitat Politecnica de Catalunya.
Moltes gracies.

And lastly to Giovanny Sanchez Rivera with whom I mostly worked with, and who made my stay in Barcelona even more vivid experience.
Muchas gracias.

Abstract

Research in the field of simulating large-scale spiking neural networks (SNN) has been carried out within the frame of Perplexus a European-funded research project based on a university consortium. In this project, a semi-custom electronic device called Ubichip has been designed. The mode of interest of this chip to emulate SNNs is based on a SIMD (Single-Instruction Multiple-Data) multiprocessor machine. The software for generating the assembly containing simulation of Iglesias-Villa spiking neural network model was also developed within that project and it is currently being successfully used for running neural network emulation on Ubichip.

The tools developed so far are useful for debugging by simulation, but in order to evaluate the behavior of SNN being emulated, two needs arose: real-time monitoring of the network evolution and a higher-level, understandable visualization solution. First, the existing software that was developed in the Perplexus project has been analyzed. After examining all available solutions, including writing a standalone dedicated program, it was finally decided to develop the so-called Ubiplot plug-in. The reason was to take advantage of the existing Ubimanager environment. The development started by verifying the communication with the Ubichip, so simple waveforms for data in a given address in the Ubichip's RAM were implemented. Then the plug-in was extended with histogram and raster plots that are accessing multiple locations of the memory in each execution step. This led to the creation of the variable map that defines the program's variables and their precise placement in the RAM. At the end simple logging facility and possibility to save and restore the layout of the plots were added.

This thesis describes the Ubiplot and the development effort put in its creation.

Contents

Chapter 1

Introduction

In modern days science tries to increase the level of knowledge in already exhaustively researched fields as well as in the evolving one or experiencing a specific resurgence of interest. That is the case with neural networks. In the turn of the nineteenth and twentieth century the neural networks became a target for a new wave of studies hoping to find new domains for theirs usages.

Neural networks are mathematical models inspired by biological nervous systems. The interconnected neurons process the information based on learning. This stands in the opposition to traditional CPUs which execute precisely specified set of instructions. Instead the network dynamically adapts and responds dependently on previous processed information. This makes them suitable for tasks very difficult to fulfill by conventional microprocessors. Neural networks found the usage in the fields of pattern recognition, data processing, studies on artificial intelligence.

As mentioned before they are inspired by the way the brain is built and works. It seems that, because of its complexity, for many years to come the human brain will remain impossible to emulate processing unit. But still creating and researching neural networks can bring us closer to the understanding of its inner-workings. It can help in diagnosing and healing the diseases and disorders of the neural system in progressively understanding it, and in implementing some simple perception functions. And maybe even in the future to fully reverse engineer brain.

There is an ongoing research in the field of simulating neural networks that was spawned by Perplexus project. Perplexus was a European Commission-funded research project. It aimed to develop hardware platform endowed with bio-inspired capabilities and to use it in studying complex behaviours in a large scale, biologically-inspired neural network. Since the hardware architecture is inspired by the real nervous systems the main interest of the project are spiking neural networks which are the closest ones to the real biological processes.

1.1 Objectives

The objective of my work was to create a solution that addresses the need of observing current state and change in the time domain of the variables of spiking neural network models being emulated by the Ubichip - a dedicated hardware device for such tasks.

The main goals of this project were to communicate with the hardware emulating the spiking neural network (SNN), and to provide three kinds of charts - a waveform, histogram and raster plots. As well as to deliver some data logging feature. One of the tasks to perform, was the choice of the possible approaches to way of the communication with Ubichip and used technology. The solution should allow to use it on Windows operating system.

The last, but not the least significant objective, was for the eventual application to be user-friendly and still functionally complete.

1.2 Spiking Neural Networks

In the structure of neural networks there are interconnected units that receive the stimuli, called action potential, that can be propagated further depending on some conditions. Using neuroscience terminology this units are called neurons and the connections - synapses.

The aforementioned condition for spiking neural networks is the membrane potential threshold. What it means is that neurons do not fire at each propagation cycle but only when membrane potential of a neuron reaches some value. The effect of the crossing of the threshold is a spike (action potential). During it the potential increase to the peak value and then decrease to its resting value. The spike is projected via synapses to all connected neurons causing change in theirs membrane potentials.

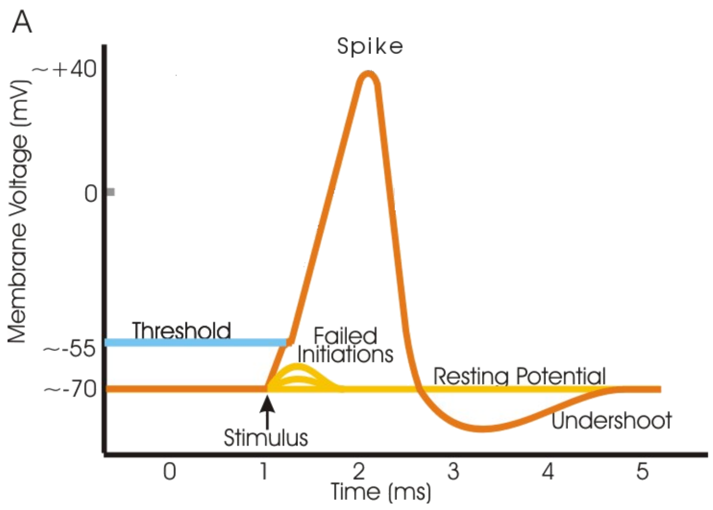


Figure 1.1: Phases of a spike[2]

A large variety of mathematical models of neural network had been created. Each of them used under different conditions and bringing different level of realism in the simulation. The most accurate to the inner-workings of the biological neuron is a HodgkinHuxley model. But the exactness of it is occupied with high computation cost. That is why it is not used for emulating big networks like the ones targeted by the Perplexus project. The one used by the project is a spiking neuron model, in particular the model created by Iglesias and Villa. This model has been analyzed and it has been made suitable to work on Ubichip.

Characteristics of Iglesias-Villa model

Iglesias and Villa falls into category of models of spiking neural networks.

At each time step the membrane potential of a neuron is calculated from formula 1.1.

$$V_i(t+1) = V_{rest[q]} + B_i(t) + (1 - s_i(t)) * (V_i(t) - V_{rest[q]}) k_{mem[q]} + \sum_j w_{ji}(t) \quad (1.1)$$

where $V_i(t+1)$ is a membrane potential of neuron type [q], $B_i(t)$ background activity, $s_i(t)$ the state of output spikes, $k_{mem[q]} = e^{-1/\tau_{mem[q]}}$ the time constant associated with the leakage of current and $w_{ji}(t)$ is a post-synaptic potentials of the jth neuron projecting to the ith neuron.

When the threshold value of membrane potential $\theta_{[q]}$ is reached neuron generates spike.

$$s_i(t) = \begin{cases} 0 & : V_i(t) - \theta_{[q]_i} < 0 \\ 1 & : V_i(t) - \theta_{[q]_i} \geq 0 \end{cases} \tag{1.2}$$

Equation 1.3 defines the synaptic weight of synapse. This variable determines the amplitude of the post-synaptic response to an incoming action potential.

$$w_{ji}(t+1) = s_j(t) * A_{ji}(t) * P_{[q_j,p_i]} \tag{1.3}$$

Where existence of a spike depends on $s_j(t)$, $P_{[q_j,q_i]}$ is the post-synaptic weight of synapse and $A_{ji}(t)$ is the activation level. There are 2 types of synapses:

- excitatory - when neuron of Type I project to neuron of Type I or II - $P_{[1,1]}$ or $P_{[1,2]}$
- inhibitory - when neuron of Type II project to neuron of Type I or II - $P_{[2,1]}$ or $P_{[2,2]}$

Type I of neurons are the ones which make excitatory connections while Type II make inhibitory. Type of the neurons is defined as a preliminary setting. Perplexus project defines the proportion of the neurons of each type on the level of 80% for excitatory to 20% for inhibitory. And the Gaussian density function describes the distribution with the σ parameter set to 10 for neurons of Type I and 75 for neurons of Type II.

The activation level used in the last formula reflects the activity of a synapse. It has got 4 states numbered as follows: $0, 1, 2, 4$. The state $0th$ indicates that synapse is not active where synapse in $4th$ state will increase neuron's membrane potential the most.

$$A_{ji}(Lji) = \begin{cases} 0 & : (A_{ji} = 1) \wedge (L_{ji} < L_{min}) \\ 1 & : [(A_{ji} = 0) \wedge (L_{ji} > L_{max})] \vee [(A_{ji} = 2) \wedge (L_{ji} < L_{min})] \\ 2 & : [(A_{ji} = 1) \wedge (L_{ji} > L_{max})] \vee [(A_{ji} = 3) \wedge (L_{ji} < L_{min})] \\ 4 & : [(A_{ji} = 2) \wedge (L_{ji} > L_{max})] \vee [(A_{ji} = 3) \wedge (L_{ji} < L_{min})] \\ A_{ji} & : (L_{ji} \geq L_{min}) \wedge (L_{ji} \leq L_{max}) \end{cases} \tag{1.4}$$

The L_{max} and L_{min} are boundaries defined by user. The activation level strongly depends on L_{ji} which is so called real-valued variable. It implements spike-timing-dependent synaptic plasticity and is defined by equation 1.5.

$$L_{ji}(t+1) = L_{ji}(t) * k_{act[q_j,q_i]} + s_i(t) * M_j(t) - s_j(t) * M_i(t) \tag{1.5}$$

Where M_i and M_j are memory of the latest inter-spike interval.

$$M_i(t+1) = s_i(t) * M_{max[q_i]} + (1 - s_i(t)) * M_i(t) * k_{syn[q_i]} \tag{1.6}$$

M_i stands for the memory of the latest presynaptic spike.

$$M_j(t+1) = s_j(t) * M_{max[q_j]} + (1 - s_j(t)) * M_j(t) * k_{syn[q_j]} \tag{1.7}$$

M_j is the memory of the latest post-synaptic spike and refers to the projected neuron. The memories are being set to its maximum values when spikes occur but when no spikes are projected they will decayed by the synaptic plasticity time constant k_{syn}.

By the time of developing Ubiplot the Iglesias-Villa model was the only available model, used by the Perplexus project. But there is also ongoing work to introduce others namely much simpler Izhikevich model. That is why the plug-in is intended to be a flexible tool suited to work with different models.

1.3 Ubichip

The primary achievement of the Perplexus project is the Ubichip. Ubichip is a flexible, reconfigurable device capable of implementing bio-inspired mechanism such as growth, learning and evolution.[3] It is a customizable hardware for emulating neural networks that thanks to its inherent parallelism has tremendous gain over the software solutions.

The target implementation is to emulate network of thousands of neurons. Each of the Ubichip is supposed to model 100 neurons with maximum of 300 synapses per neuron. And the series of Ubichips will be connected with each other, propagating spikes using a hardwired AER (Address Event Representation) bus, the Ethernet or wi-fi.

Ubichip works within Ubidule that is a board which purpose is to make communication and controlling of Ubichip possible. It is equipped with Colibri board with high-end XScale embedded processor. It runs Linux operating system.

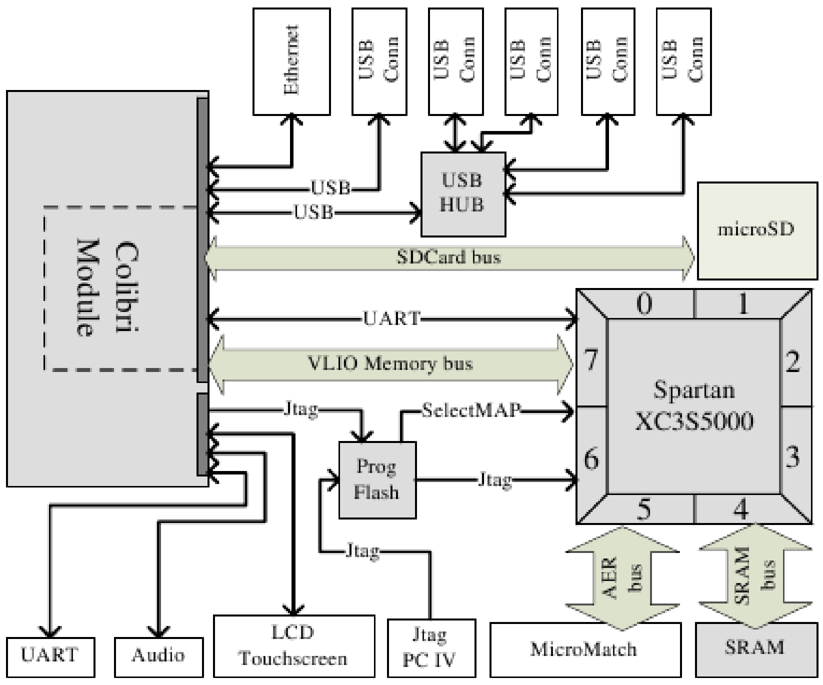


Figure 1.2: Ubidule schema

The building blocks of Ubichip seen in the Figure 1.3 are as follows:

- Macrocell - consists of 4 4-bit ubicells, dynamic reconfiguration unit and self replication unit. A Macrocell features a 16-bit Processing Element (PE) that is the basic element used to emulate the SNNs.
- Macrocell array - an array of Macrocells.
- Configuration unit - it configures the Ubichip's blocks
- Memory controller - communicates with external RAM
- Sequencer - fetches and decodes the instructions stored in the memory then broadcast them to the array of PEs.

For running spiking neural network's Ubichip is configured in Single Instruction Multiple Data (SIMD) mode. In this mode each of the aforementioned Ubicells in the Macrocell implements a 4-bit ALU. The 16-bit PE (or Macrocell) consists of 4 Ubicells. All of them process synchronously the same

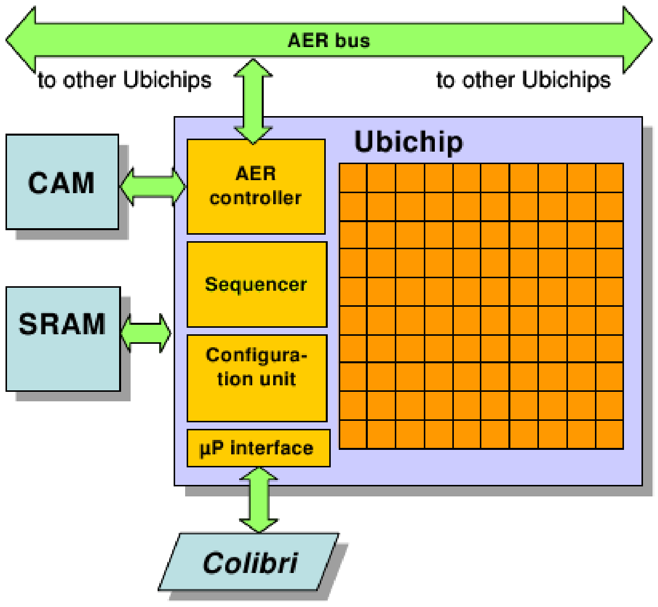


Figure 1.3: Ubichip schema

instruction but with different data. The execution is governed by the sequencer. each of the Macrocells is regarded as a single neuron and a single processing element (PE) for the sequencer.

The execution on Ubichip can be divided in two phases. In the first one the code is executed by the PEs until the STOP instruction has been encountered. Then second phase begin. In it the sequencer is disabled and control is given to the CAM controller. The Content Addressable Memory (CAM) stores the information about connectivity of the network. The CAM controller scan the array in search of fired neurons. When such are found the spikes are being propagated to appropriate neurons.

1.4 Spiking Network Development Kit

The Spiking Network Development Kit (SpiNDeK) was created by Michael Hauptvogel[6] and extended by Marc Hortas[7]. It is a desktop application that generates assembly for Ubichip. The assembler code includes the implementation of the Iglesias-Villa model. The initial values of the variables are listed in the appendix C. To overcome the 16 bit limitation of the Processing Element of Ubichip the program operates on integer types instead of floats. SpiNDeK also generates the memory map with variables placements that is loaded to Ubichip's RAM.

Important from the point view of the visualisation software is the memory structure. The map for 100 neurons with 300 synapses each is shown as an example in the Figure 1.4.

To pack data effectively in one 32 bits row there are two 16 bits sections with the variables. These sections are SP1, SP2 for synapse variables and NP1, NP2, NP3, NP4 for neuron variables as shown in the Figure 1.5

The first 317 addresses depicted in the map are pointers. Next there is block of instructions, then block of synapse parameters and the block of neuron parameters. The last block holds common variables.

The pointer at address 0 is used as PC. Each of the pointer from address 1 to address 300 point to a block of synapse parameters. The block is built in such way that it holds variables from SP1 and SP2 sections of the *ith* synapse of every neuron. Next pointer at address 301 points to the block of neuron variables from NP1 sections of each neuron. Pointers under 302 and 303 defines position of variables from NP2 and NP3 sections in the same manner. Data in addresses from 304 to 317 hold addresses for common variables. Details of all the parameters can be found in [6].

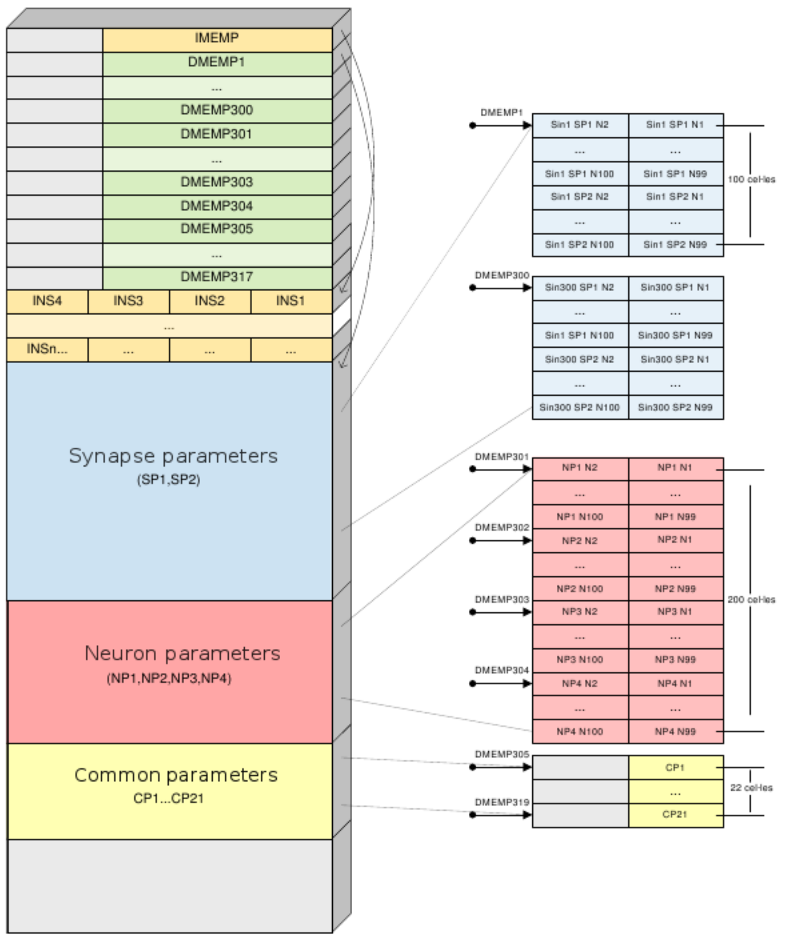


Figure 1.4: Memory map generated by SpiNDeK for 100 neurons with 300 synapses each.

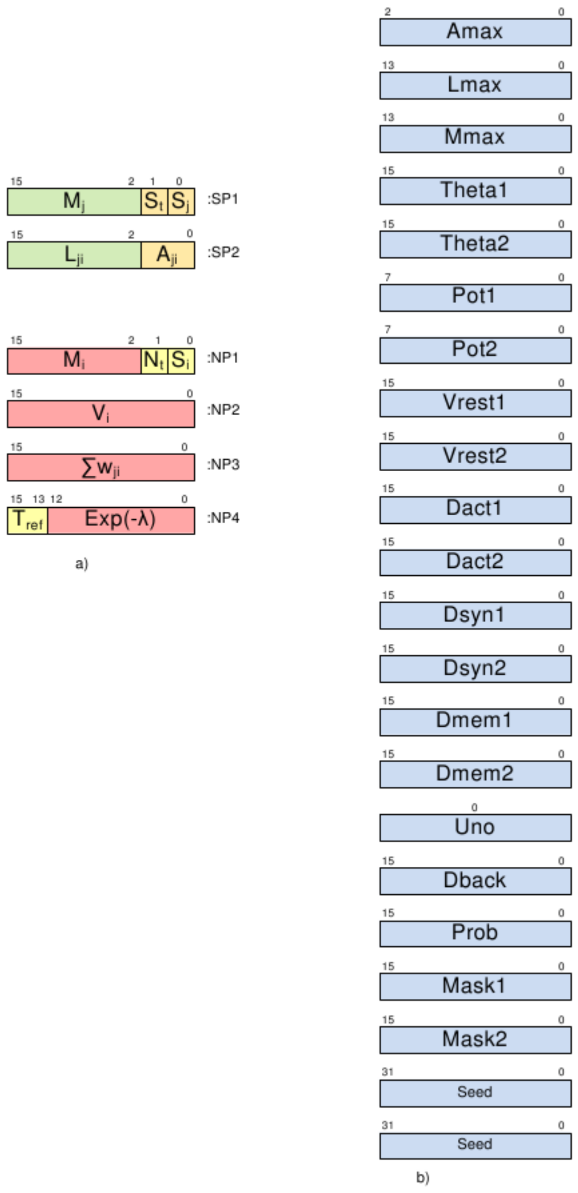


Figure 1.5: Variables' placement within memory row.

Chapter 2

Implementation details

In this chapter firstly the existing software created for work with Ubichip will be presented and its influence on the project. Later it will focus on the implementation strategies, the structure and collaboration between Ubiplot's classes. It can proof useful for anyone wanting to alter or extend the plug-in with some new features. It should be noted that adding new kind of chart to Ubiplot as an example is shown in the User's guide appendix (section A.3). Also this chapter will not elaborate on the implementation in depth it will rather give valuable overview for understanding the source code. It will mention used Qt's classes[8] namely the container classes, QWidget, QTimer, QPaint so some basic knowledge of them and Qt is advised.

2.1 Software base

The software stack called Ubimanagertools was developed for work with Ubichip - to configure the hardware, manage the simulation and more. This programs were developed using Qt framework. Notably two of the application written as a part of it was a base for Ubiplot development - Ubicolibri, the server that runs on the Ubidule and Ubimanager.

Ubimanager is application that connects to this server by means of an Ethernet connection. Its purpose is to control, observe simulation and to configure it as well. It instructs server to do variety of things, execute the code, alter PE array, read and write to RAM. It has got also a plug-in facility, which means it can load dynamically during runtime code written by 3rd party.

These plug-ins can benefit from the interface between Ubidule and Ubimanager. That means that no extra effort has to be taken in establishing the communication between board and user's PC. In the planning phase of the development of Ubiplot few scenarios were taken under consideration. One was to write a server that would run on the board and a desktop application for the PC. But that server would essentially duplicate the functionality of Ubicolibri. Also, Ubimanager guarantees that in the future when the hardware would change the Ubimanager's interface for plug-in would stay the same keeping the already written software working. For that reason creating plug-in for Ubimanager was chosen eventually, which among other mentioned benefits had also cut the development time significantly.

Deciding on writing the plug-in results in the necessity of using the Qt framework. Although it could be possible to write Ubiplot in python or ruby that are supported by Qt, but C++ as a "native" framework's language seemed the less cumbersome choice and such was taken.

2.1.1 Qt

Qt is a cross-platform development framework[8] primarily for GUI but also for console application. It is written in C and C++. The creator of Qt was company named Trolltech that later changed its name to QtSoftware and now is a part of Nokia corporation. It is licensed under the terms of the GNU Lesser General Public License (LGPL).

Qt supports wide range of platforms, namely: GNU/Linux, Microsoft Windows, Mac OS X and embedded operating systems Windows CE, Symbian, Embedded Linux QWS. The GCC compiler is a main supported compiler, but Qt programs can be compiled with Visual and Intel compilers among others.

From the user's point of view the framework consists of:

- qmake - the building tool
- moc - the C++ preprocessor
- library - rich API

qmake is tool that aids programmer in the build stage of the application. It reads the project file (the one with the pro extension) in the directory with source files and creates makefile accordingly. The project file can have numeral options that influence the generated makefile - whether the software is a plug-in, library or ordinary executable program, to include debugging symbols in the target file, external libraries to link to and many others. The makefile thus created also invokes the moc program.

moc is a preprocessor used in the early compilation phase. It reads source files and looks for Qt macros and is using them to generate additional code that provide extra features not available in native C++. These are signal/slot mechanism, introspection and asynchronous function calls.

Ubiplot is written using common Qt classes like QString, QVector<>, QMap<>(Java style containers), QObject.

QScriptEngine an engine for ACMEScript (commonly known as JavaScript) was clean and simple solution for creating variables memory map parser. It is used in the ScriptEngine class.

Ubiplot also uses signals/slots mechanism, and Ubimanager plug-in infrastructure is based on Qt's introspection.

2.1.2 Ubimanager interface

Plug-in have to inherit from UbiSimplePlugin class. This class implements

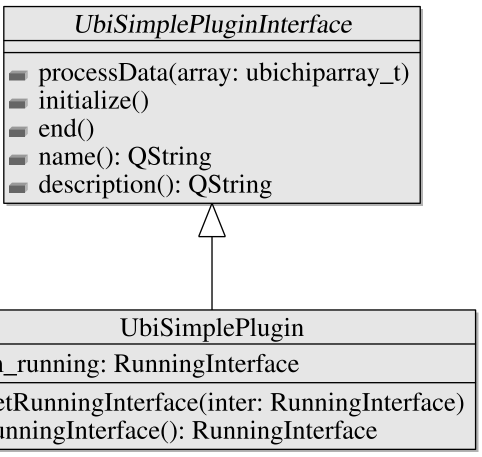

UbiSimplePluginInterface
processData(array: ubichiparray_t)
initialize()
end()
name(): QString
description(): QString
UbiSimplePlugin
m_running: RunningInterface
setRunningInterface(inter: RunningInterface)
runningInterface(): RunningInterface

Figure 2.1: UML representing UbiSimplePlugin class

UbiSimplePluginInterface which has two pure virtual methods: name() and description(). These methods have to be implemented in the plug-in as they are used by Ubimanager to probe for name and description. The initilize() and end() are methods called when loading and unloading the plug-in.

The method processData() is called every time new data from server have arrived. That is when the Ubimanager establish connection with the server, execution of steps has finish, break or reset command have been instructed by user. In the body of this method plug-in can access the running simulation using data types defined in ubitypes.h header (taken from Ubimanager's source code). It accomplishes it by the means of the RunningInterace* m_running variable.

Some of the methods of RunningInterface that are used by Ubiplot:

- Run() - used to instruct board to do continuous run.
- rBreak() - used to instruct board to execute break.
- ramRead() - used to read RAM memory (see SRAM in Fig. 1.3) from specified address (it is blocking operation).
- ramWrite() - used to write RAM memory to specified address.
- chipRead() - used to read from chip (see Ubichip in Fig. 1.3) from specified address (it is blocking operation).
- chipWrite() - used to write to chip to specified address.

The methods for accessing the SRAM and Ubichip accepts the parameters for specifying the RAM/chip location and the variable's pointer to read to or to store the data. The first kind of parameters consists of address, number of rows (nb) and the delta (delta). The call with default parameters (nb = 1, delta = 1) will result in accessing single row. The nb greater then 1 will result in accessing more than one row. And the value of delta tells server what is a difference between the current address and next one while accessing multiple rows.

As stated in the list reading methods are blocking - the execution of whole application is blocked until the server responds to them. After call to ramRead() and chipRead() the content of RAM can be extracted from UbiSRAM object that is populated with requested data. The object is retrieved using call:

runningInterface()->getUbichipArray()->ubichip()->ram

The Ubicolibri code for this methods is the same code as used by ubichiptester which can be found in the Ubimanagertools. And is used on the Ubidule for accessing the RAM and chip.

2.2 Overview

When Ubimanager loads the plug-in, it creates an instance of PlotPlugin. Next it calls PlotPlugin::initialize(). In this method PlotWindow is being created and stored as a class variable. It is a main window of the plug-in and an interface for creating plots, controlling the simulation and logging data.

2.2.1 Operation principle

The assumption is that the code executed on the Ubichip is going into STOP state right after execution cycle finishes (when the control is given back to

CAM controller by means of a STOP instruction). When CAM has control the computation of variable have finished and the propagation of spikes occur. It is an appropriate time for prompting the server for data. The state of the sequencer is stored in RAM. 1 means it is stopped 0 otherwise. By classical polling the plug-in is checking the state of sequencer. If it is in halt if fetches the data and escapes the halt by writing value 1 to the location in the chip.

2.2.2 Handling Ubimanager interface

PlotPlugin is the name of the Ubiplot's class that inherits from UbiSimplePlugin. Although the intention of the authors of the Ubimanager was for plug-ins to use UbiSimplePlugin::processData() method for handling the data from the server, Ubiplot does not make use of it. This is because as explained in the section 2.2.1 Ubiplot is using a polling technique to check if it is the right moment to ask for data. PlotPlugin class has got two slot methods in which it is communicating with server:

- stepPoll()
- runPoll()

The QTimer object is being connected to them. To which of these methods the QTimer will be connected depends on what type of execution user selected in the main window. If single step - stepPoll() - if run was chosen runPoll() will be used. They accomplish the same tasks. First they check the halt flag. If it is set they ask UbiSimplePlugin::RunningInterface object for ram read. Then they escape the halt. The difference between them is that stepPoll() stops the timer since only one execution step was meant.

The parameters for the call in polling methods to RunningInterface::ramRead() are taken from the DataRequest object of every PlotWidget. More can be read about DataRequest later in this chapter in the section 2.3. The most obvious strategy would be to ask PlotWindow instance for DataRequest objects. But since the execution of the polling methods have to be vary fast the number of functions calls in them had to be minimized. So the PlotPlugin is storing QVector object with pointers to DataRequest instances. The new elements to this vector are being appended when new plot is created. There is connection between PlotWindow::newDataRequest(DataRequest*) signal and PlotPlugin::newDataRequestSlot(DataRequest*) slot that serves this purpose.

2.2.3 Inheritance of the main classes

Classes responsible for creating the plots are:

- PlotWidget - QWidget that is a parent for every other object in this list. From PlotWindow perspective it encapsulates whole plot.
- Plot - QWidget that have QWidget::paintEvent() overloaded. In that method the painting of data is taking place.
- PlotData - QObject that incoming data is passed to and stored. Used by Plot to draw the data, LogWindow to log the data.
- DataRequest - object that represents the request for data - address and other arguments, more on this can be found in the section 2.3.

The collaboration between them is depicted in the Figure 2.2. The tasks of each class is simple. PlotWidget holds all objects of other classes. Plot is used for painting. PlotData is saving the sample and DataRequest holds the information needed for a request for RunningInterface.

The plot is created in the manner that first the DataRequest is created using variable map (see subsection "Variables menu" in A.1.1) or by manually specifying the address. Then PlotWidget is being created. Every needed information is passed to its constructor. In the constructor appropriate Plot and PlotData are instantiated. The last step is to call setter PlotWidget::setDataRequest(DataRequest*). It sets the created before DataRequest and also connects DataRequest::newData(unsigned short int*) signal to PlotWidget::newDataSlot(unsigned short int*) slot. Via this connection the fetched data is passed from server. Moreover it also sets the widget as a parent for the DataRequest object so it will be destroyed with it.

Ubiplot supports three kinds of charts. For every kind there is equivalent derived class. Generally the names of these classes are constructed by adding prefix to the name of the parent. Although with children of PlotData class there are exceptions.

Later in this chapter each of this classes will be presented. Every time class specialised for the raster plot will be the first one since it is the most straightforward and conceive.

2.3 DataRequest class

Objects of this class encapsulates the request for data and manipulation of it before serving to PlotWidget objects. Most important methods of DataRequest are:

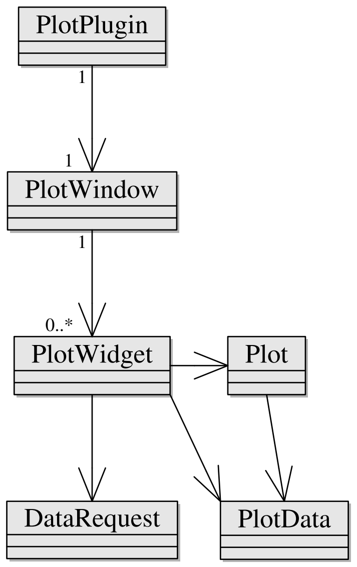


Figure 2.2: UML Figure shows collaboration between classes responsible for plotting.

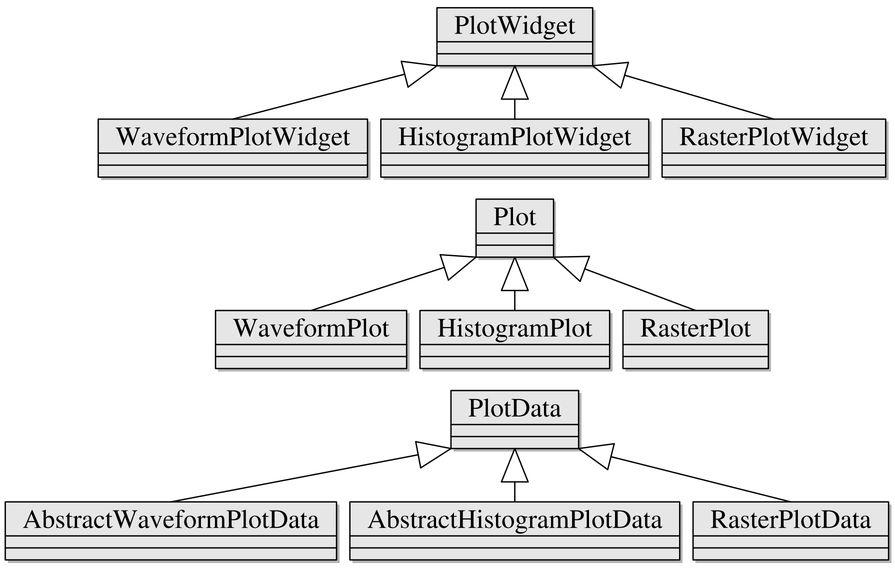


Figure 2.3: Family of the plotting classes

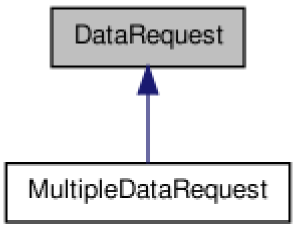


Figure 2.4: Inheritance of DataRequest class.

- bool respond(UbiSRAM &) - the downloaded data is passed to it and is being masked out.
- void newData(unsigned short int*) - signal that is being emitted to PlotWidget with new data.

In the polling methods of PlotPlugin the respond() method is called. The request can be made for a pointer or directly to an address. If the call to respond() returns false the PlotPlugin will recall it second time. In that case it means that this is request for pointer. So first call to respond() will derefence the pointer and second one will passed the data on. response() method also is masking out the memory row.

As seen in the Figure 2.4 there are two classes - DataRequest and MultipleDataRequest that inherits from it. Simple DataRequest is used when plot has to access single location in the memory, like in the case of a waveform type. But since histogram and raster plots needs more than one value each execution step they are using MultipleDataRequest. It holds DataRequest objects in the QVector<DataRequest* >. The respond method of it is using the same code as DataRequest::respond() to copy data from UbiSRAM instance and then it is emitting newData().

2.4 Storing data

Plot's data is stored in the object of classes derived from PlotData. PlotData is an abstract class. It constitutes following interface:

- void newData(unsigned short int*) - new data is passed through this variable.
- void toYML(YAML::Node) - writes the state of its variables to yaml format.

- QString toHtml() - returns html table filled with stored data, it is used by LogWindow object.
- QString desc() - returns short description about itself used for updating the PlotWindow status bar.

The primary method is newData(). Through it the data is passed (from an object of DataRequest) to the plot.

Every derived class stores the data in different container and before saving it manipulates it some way appropriate for the type of the plot. If the container's length exceeds the value of "loggingLimit" variable (defined in PlotData) it is truncated accordingly.

While creating this classes it had to be taken under the account that except for raster plot the rest can be interpreted data as coded simple or twos complement. That is why there are template classes for histogram and waveform plots. While developing the plug-in template classes were the obvious choice but they got two disadvantages.

One of it is that the moc preprocessor does not work with them so there are no signal/slot mechanism. To tackle this problem first the approach with boolean variable indicating used code and C macros was tried. The source code thus written was not elegant and very hard to maintain. But since it was possible to boil down the signals emitting to only one by WaveformPlotData this problem faded away.

The second "con" against template classes is that it would change the simple composition of the classes depicted in the Figure 2.2 because PlotWidget and Plot would have to store pointer to two objects of PlotData. The workaround for this problem was the introduction of an AbstractWaveformPlotData and AbstractHistogramPlotData. They are non template classes that adequate PlotData classes derive from. In this way only pointer to these classes is stored in the WaveformPlot and HistogramPlot but the pointer actually points to the object of a template classes.

2.4.1 Raster plots

The simplest approach of the inheritance of PlotData was taken to the RasterPlotData.

To the newData() method pointer to an array of unsigned short ints is passed. Because raster plots shows many values for each execution steps that is why (in opposition to the waveform's PlotData) this array is bigger than 1 element. It is of size equal to the "values" variable of RasterPlotData. In order to save the data the array is being iterated and the data is being stored

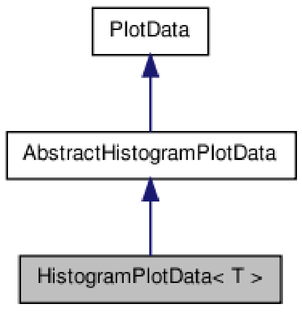


Figure 2.5: HistogramPlotData inheritance graph.

in the QVector<bool* >. It should be read as follows, the vector is storing arrays of boolean values. Each array reflects the passed data for each of the execution step.

The toHtml() method creates html table. The columns of the table are execution steps and rows represents each values on the y axis of the plot.

2.4.2 Histogram plots

More complicated inheritance can be observed when viewing the PlotData for histogram plots. It is depicted in the Figure 2.5

Before the HistogramPlotData there is AbstractHistogramPlotData that hides the templateness when storing pointer to it in the HistogramPlot object. This approach was discussed earlier in the section 2.4.

HistogramPlotData stores the data in the map QMap<float, unsigned int>. As with waveform counterpart the data passed to the objects of this class can be interpreted as twos complement or simple coded. The keys of this map represents the ranges and values the number of occurrences. Each key is the inclusive beginning of the range:

$$key_i \leq sample < key_{i+1}$$

The number of entries in the map is equal to resolution defined by the user upon creation of the plot. The values parameter passed to the constructor indicates how big the array passed via newData() will be. In this method filling of the map is taking place.

The creation of the keys is drafted in pseudo code in algorithm 1. Keys are calculated using the extremes so they will cover all of the possible values. The implementation differs with this pseudo code in the way that all of the values of the map are set to 0, except for the first one which is set to 1

Algorithm 1 Pseudo code for filling the map

```
sort(data)
min = data[0]
max = data[1]
repeat
    step = abs(max-min)/resolution
    map.insert(min + step*i, 0)
    i = i + 1
until $i < data.size$ and $max \neq min$
```

because it is known that there will be at least one value in this range - the minimum.

The pseudo code that matches values to ranges is shown in algorithm 2. In the implementation index variable *dataIndex* is set to 1 to omit the first

Algorithm 2 Pseudo code for filling the map with keys, step is a variable from previous algorithm, values is a size of the passed array of data

```
keyIndex = 0
dataIndex = 0
while $dataIndex < values$ do
    key = map.keys[keyIndex]
    if $map[key] \leq data[dataIndex] < data[dataIndex] + step$ then
        map[key]++
        dataIndex++
    else
        keyIndex++
    end if
end while
```

data since it was taken care of while creating ranges. Another improvement in the implementation for this algorithm is that when looping through the array of data if the *keyIndex* is the last index of the map it means that all the remaining elements can be assign to this last range and the execution is terminating the loop with the break instruction. The whole implementation is listed in the appendix E.

The most important features of these algorithm is that the data array is already sorted so finding the correct range is fast. If the value does not match the current range it means that every possible values for this range have been found, so it increments *keyIndex* thus omitting it in next iterations.

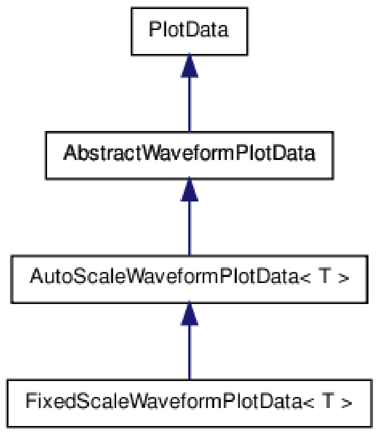

PlotData
AbstractWaveformPlotData
AutoScaleWaveformPlotData< T >
FixedScaleWaveformPlotData< T >

Figure 2.6: Inheritance of PlotData for waveform plots.

2.4.3 Waveforms plots

User can choose to create plot with automatic scale that will auto adjust its scale while new samples income. Or fixed scale and define the minimum, maximum and the desired resolution that is the height of the plot in pixels. Depending on the desired type of scale AutoScalePlotData<T>or FixedScalePlotData<T>is used.

Most of the logic is placed in common parent. AbstractWaveFormPlotData and the derived classes override newData() method. Reason for using template is the same as with HistogramPlotData - because of the two way of interpreting the data in simple or twos complement code. Data is being stored in QVector<T >.

For other kind plots the task of PlotData is to simply store the samples that will be later retrieved by an object of Plot. But since WaveformPlot object holds pointer to AbstractPlotData it does not know the used template and as oppose to HistogramPlot it would have to fetch data in container holding <unsigned short int> or <short int> data types. That is why it has got additional task - to draw the data in methods paintData() and paintLabels().

2.5 Usage of yaml format

The yaml format is used by the plug-in for two tasks: providing the serialization feature for plots and defining variables in the variables' map.

The yaml format was used in favor to xml because it is easier for humans to use it. This is most important advantage in the case of variables' map that have to be created by the user of Ubiplot.

The plug-in is linked to the yaml-cpp library for handling the yaml. It is available under the MIT license and the version 0.2.5 of it is shipped with the Ubiplot's source code. It can be found in the yaml-cpp directory.

2.5.1 Serialization

The layout of the plots can be saved to yaml file and then later restored. Classes that constitute plots have got toYML() methods. These methods saves the state of variables needed to later recreate the Plot. To deserialize objects static methods fromYML() are used. They parse the yaml nodes and returns instantiated objects.

2.5.2 Implementing variable's map feature

As described in section A.1.2 it is possible to use variable's map to create a menu in the plug-in's main window. By triggering one of the menu's entry the plot for specified variable of neuron or synapse will be created.

This is accomplished by the ScriptEngine class. It is a wrapper around QScriptEngine object. It parses the yaml file and stores information that defines a variable in the VariablePath objects. PlotWindow while creating the menu asks ScriptEngine for names of the neuron and synapse variables and it populates its menu accordingly. Then it connects QActions in the menu to appropriate slots. These slots call ScriptEngine's createDataRequest() or createMultipleRequest().

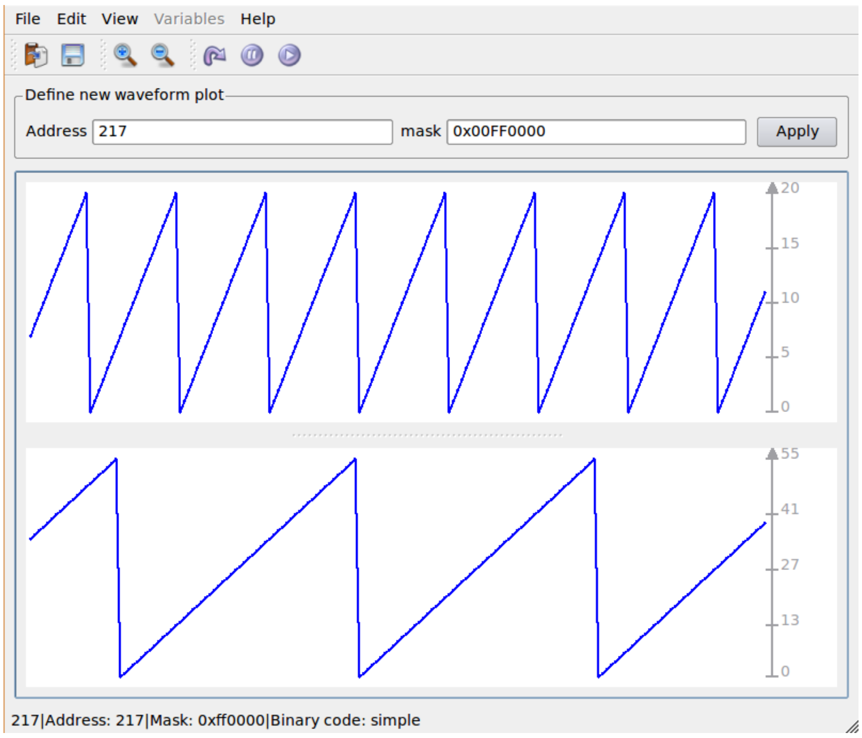
Chapter 3

Results

In this chapter experimental results will presented using available facilities of Ubiplot. First simple program will be used to test waveform plots and communication with the board. And next some variables of the SpiNDeK's implementation of Iglesias-Villa model will be plotted.

Manually specified address

The program that simply increases two variables in the RAM had been loaded to the Ubichip. If one wants to observe these counters it is possible to manually specify the address and mask for RAM's request. The result is depicted in the Figure 3.1.



217|Address: 217|Mask: 0xff0000|Binary code: simple

Figure 3.1: The plug-in plotting two counters.

3.1 Visualisation of Iglesias-Villa model

Now the program simulating Iglesias-Villa model will be the subject of the observation. The array is 4 neurons with 3 synapses each. The connectivity created by SpiNDeK is depicted in the figure 3.2. The whole program that Ubichip was executing is listed in the appendix D.

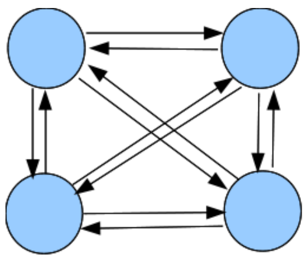


Figure 3.2: The connectivity of the observed network.

Raster plots

In the Figure 3.3 raster plot is used to check the types of the neuron. The plot is showing variable N_t that is holding the type of the neuron, 1 stands for excitatory and 0 for inhibitory.

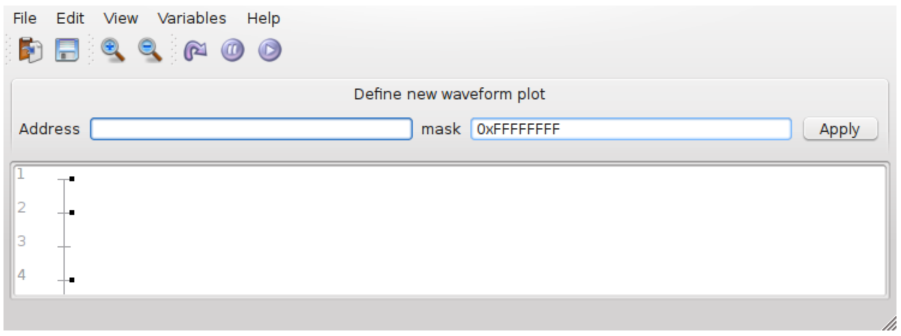
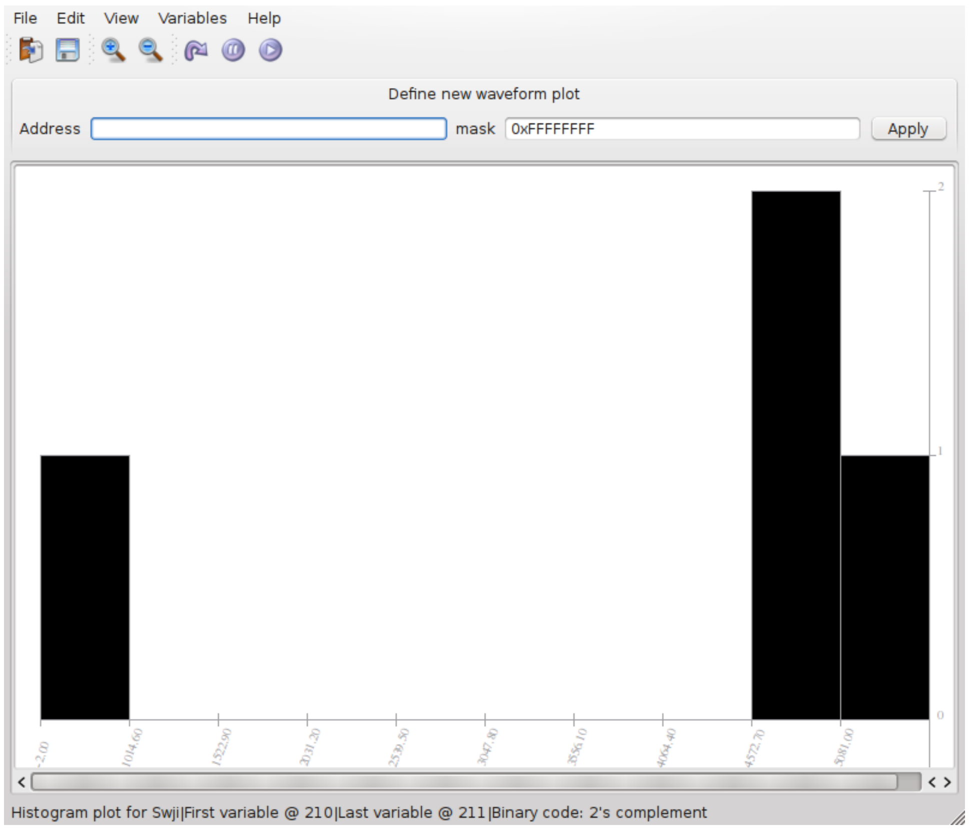


Figure 3.3: Raster plot showing types of the neurons.

Histogram plots

Figure 3.4 shows an example usage of the histogram plots. The plot is depicting the state of the synapse variables of all the neuron in the current step.



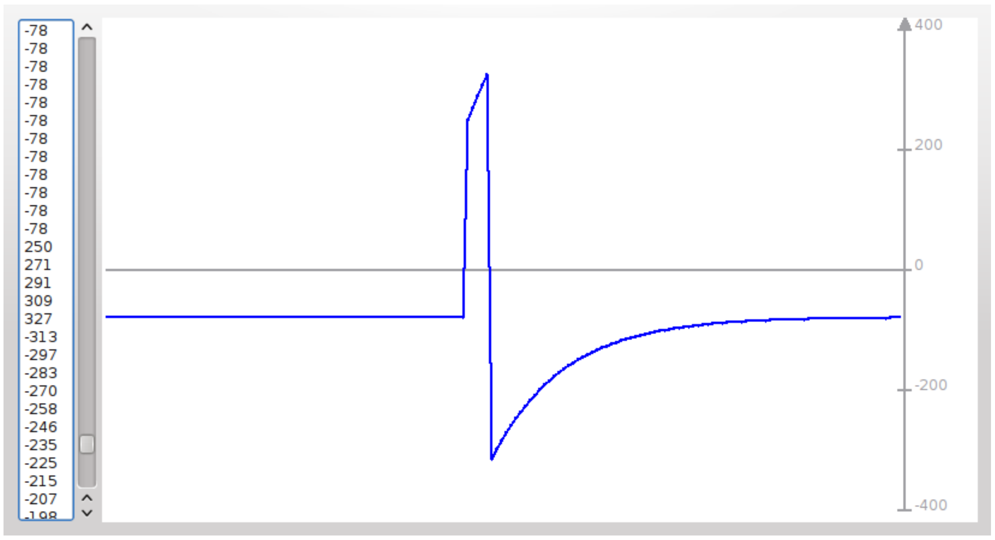
mask: 0xFFFFFFFF

Histogram plot for Swji|First variable @ 210|Last variable @ 211|Binary code: 2's complement

Figure 3.4: Histogram plot for synaptic weights. The resolution is set to 10.

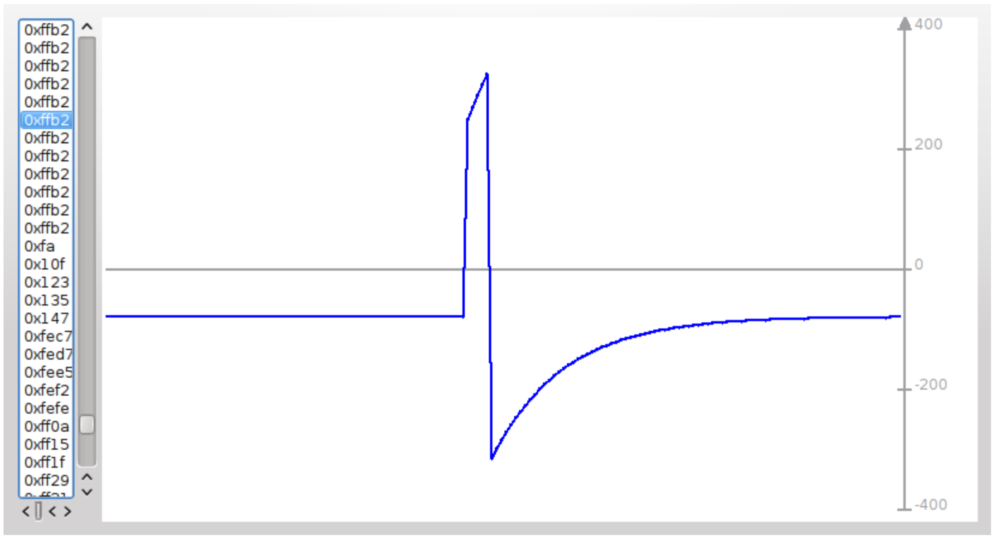
Waveform plots

The Figure 3.5 shows the plot visualising the the membrane potential of first neuron while emitting spike. The list on the left shows values for the fragment of the plot when the spike occur.



```
-78
-78
-78
-78
-78
-78
-78
-78
-78
-78
-78
-78
250
271
291
309
327
-313
-297
-283
-270
-258
-246
-235
-225
-215
-207
```

Figure 3.5: Waveform showing Vi of first neuron while spiking, list on the left shows values in the decimal base.

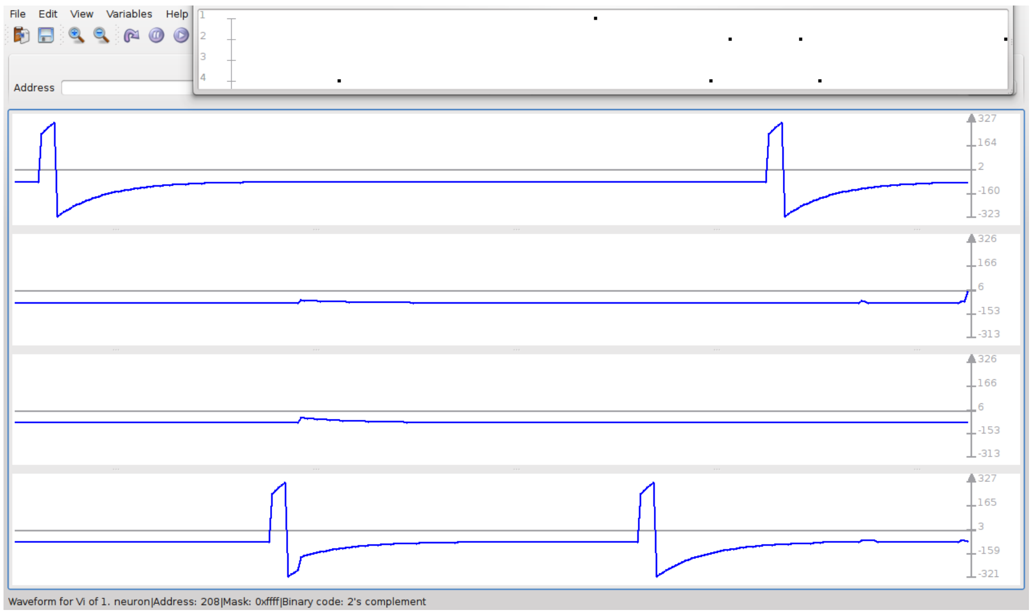


```
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xffb2
0xfa
0x10f
0x123
0x135
0x147
0xfec7
0xfed7
0xfee5
0xfef2
0xfefe
0xff0a
0xff15
0xff1f
0xff29
```

Figure 3.6: Waveform showing Vi of first neuron while spiking, list on the left shows values in the hexadecimal base.

Showing multiple plots at the same time

Ubiplot displaying four plots in the main window and one plot in the separate window is showed in the Figure 3.7. The plots in the the main window



Waveform for Vi of 1. neuron|Address: 208|Mask: 0xffff|Binary code: 2's complement

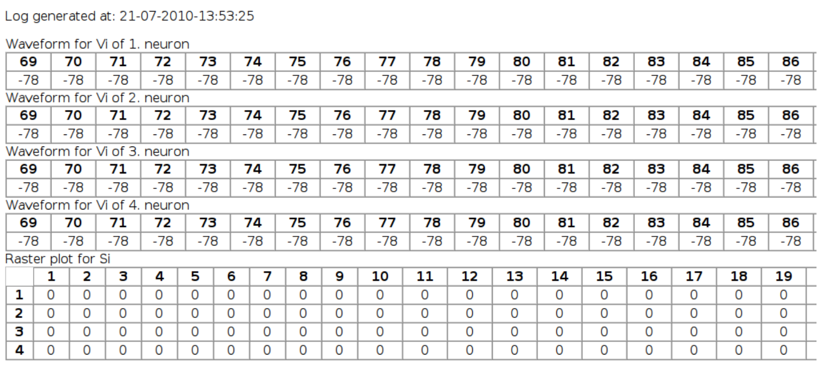
Figure 3.7: Visualising membrane potentials in the waveforms and spikes using raster plot in the separate window.

are waveforms for membrane potentials, where spikes are visualised in the separate window using raster plot.

There one can observe that the first and fourth neurons emitted spikes twice. The raster plot simultaneously shows the occurred spikes. It should be noted that part of the spikes are caused by background activity and not by neurons. It is necessary to keep the network working.

Logging data

The part of the logs in the html format of this session is attached in the Figure 3.8. The headers are the number of steps. They start from 69 since the limit for logs was set to 1000 and 1069 steps were executed.



Log generated at: 21-07-2010-13:53:25

Waveform for Vi of 1. neuron

69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86
-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78

Waveform for Vi of 2. neuron

69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86
-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78

Waveform for Vi of 3. neuron

69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86
-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78

Waveform for Vi of 4. neuron

69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86
-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78	-78

Raster plot for Si

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.8: Part of the logs.

Chapter 4

Conclusions

At the end, the created software is being used successfully and is pushing the research forward. It is also looking promising in the context of future usages.

It does it thanks to the Ubimanagertools, namely the server and Ubimanager. Those two constituted rich interface that programmers can exploit. Although plug-ins' facility of Ubimanager was not intended for such applications, but it was possible to cooperate with by using slots triggered by QTimer for communication instead of original processData() method of UbiSimplePlugin interface. Overall software available in the Perplexus project turned out to be good base to work with.

Also the Qt framework, as expected, proved its usefulness. It is very reliable, easy to use and its API complete. Every time there was a need for some feature the ready solution that can be used or in some way altered could be found in the Qt library. That was the case when implementing variables' map feature. The only thing missing was the support for yaml format and the external yaml-cpp library have been introduced. In fact xml that is supported by Qt could have been used instead of yaml, but the readability was prioritized. And of course thanks to Qt Ubiplot is cross platform, which obviously is an advantage for end users since they can use it not only on one operating system. Moreover it was possible for development to take place in the unix userspace, which in the author's opinion is superior for such task to widely used windows environment.

During the development question concerning the used language arose. Although writing Ubicolibri and Ubichiptester in C++ was an obvious choice, since these applications run very close to hardware, in the case of Ubimanager and Ubiplot was not. I have many doubts if it was the best language to use. For sure it is very fast. But on the modern computers really the only bottleneck of Ubimanager was the operation on sockets. Other were not so crucial. It is possible that usage of other language together with bindings for

Qt, like Java (QtJambi) or Python (PyQt), would be a better choice.

That is why I think at the beginning of the development of Ubiplot more time could have been put on the research of the available technologies. Especially use of Python language that integrates easily into C, together with PyQt looks promising. Substitution C++ with less verbose and error-prone language would benefit in shorter development time and more maintainable code.

Also the decision about implementing plots had to be made. Eventually writing own solution from scratch was chosen. But as it turned out amount of time needed for that task was bigger than expected. Now, incorporating some library like for example "qwt" seems a faster and more reliable solution.

To summarize, the pursued objectives have been successfully achieved. The Ubiplot plug-in communicates correctly with hardware and displays the waveforms of the user-selected variables in three different kinds of plots, waveform, raster and histogram. Several experimental verifications have been successfully performed and reported.

Future work

As future work, replacing polling with some callback technique is suggested. That would require contacting the development team of Ubimanagertools and implementing this in Ubicolibri and Ubimanager code.

In the meantime, the way Ubiplot is checking the state of the stop flag could be made more customizable with the use of the QScriptingEngine. The engine might parse user-inputed expressions that would instruct the plug-in how to check and escape the flag, and when to fetch the data.

Another enhancement in the communication part of the Ubiplot might be replacing QTimer objects with QThread, that is starting new thread for polling process. It should give considerable boost of the performance.

Also user interface could be further enriched by giving the user possibility to save plots in the context of variables shown and not, like in the current implementation, precise addresses in the memory. That would require developing slightly more robust serialization facility than is right now.

As for other extensions they should be easy to implement since Ubiplot's structure is a straightforward and simple one, but still sufficient for future work on it.

Bibliography

[1] Report originator: Andreus Upegui *Deliverable 3.1: Specification of the computing module's main electronic board*, November 2007.

[2] Wikipedia: The Free Encyclopedia. Wikimedia Foundation Inc. Encyclopedia on-line. Available at *http://en.wikipedia.org/wiki/Action_potential.* Internet. Retrieved 20 July 2010, 20:30 UTC.

[3] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J. Moreno, and J. Madrenas, *The Perplexus bio-inspired recongurable circuit, in Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems*, T. Arslan et al, Ed. Los Alamitos, CA, USA: IEEE Computer Society, August 2007, pp. 600605.

[4] Report originator: Jordi Madrenas *"Pervasive computing framework for modeling complex virtually-unbounded systems", 18 November 2008*

[5] Lukasz Kotynia, Supervisor: J. Manuel Moreno Arostegui *"Design of CAM memories using FPGAs for implementing the encoder/decoder modules for PERPLEXUS"*

[6] Michael Hauptvogel *"Design of a bio-inspired spiking network environment"* (MSc dissertation) Universitat Politecnica de Catalunya, 17 March 2008, p. 19-34, 88

[7] Marc Hortac *"Desenvolupament de software i algorisme per a xarxes neuronals spiking inspirades"* (MSc dissertation) Universitat Politecnica de Catalunya, September 2009, p. 61-62

[8] Qt Reference documentation *http://doc.qt.nocia.com/4.6*

Appendices

Appendix A

User's guide

This chapter is a user guide. It shows how Ubiplot can be used. Beginning with some common tasks and later will show how to create a variables' map. It will also elaborate on the building process and give example how new kind of plots can be written.

A.1 Using the program

A.1.1 Common tasks

Loading the plug-in

First of all Ubimanager have to be started. From its "Simulate" menu the entry "Connect to simulation as a observer" has to be selected. Then it is possible to choose the board user wish to connect to. After connecting state of the simulation is downloaded and entry "Connect to simulation as a master" in the menu "Simulation" becomes enabled. In order to load the plug-in and have control over the simulation user have to disconnect and connect to the same board again but this time using the just enabled entry in the "Simulation" menu. While connecting the dialog is showed with loadable plug-ins that are placed in the special directory. This directory can be set from "Tools"->"Plugins directory". The directory should be the one with Ubiplot's file, on windows it is called "ubi_plotplugin.dll" and on unixes "libubi_plotplugin.so".

After loading the plug-in main window of Ubiplot will be shown.

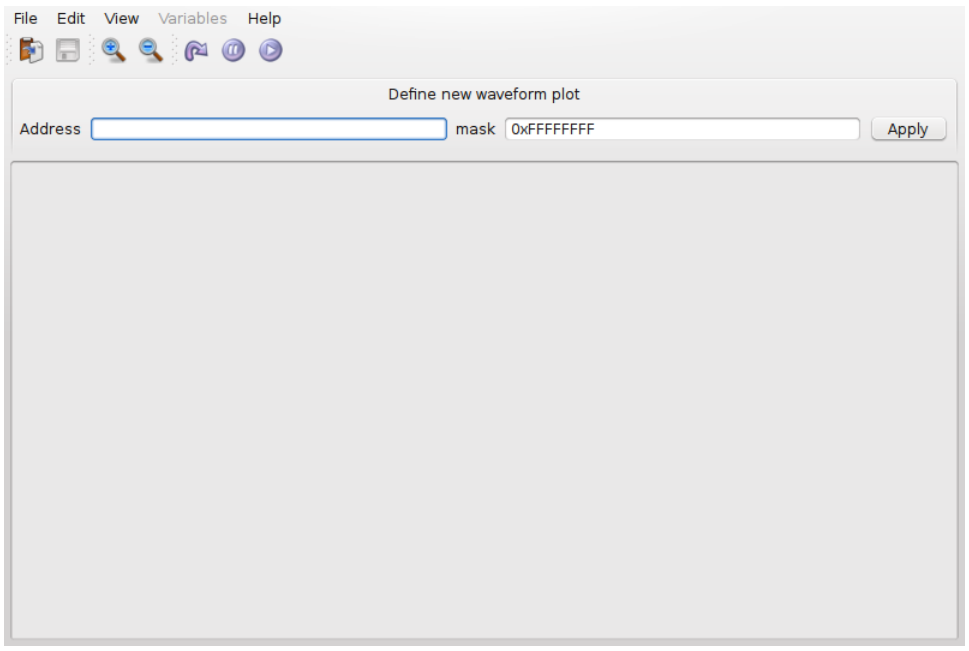


Figure A.1: Main window of Ubiplot

File menu

There user can choose from the "File" menu:

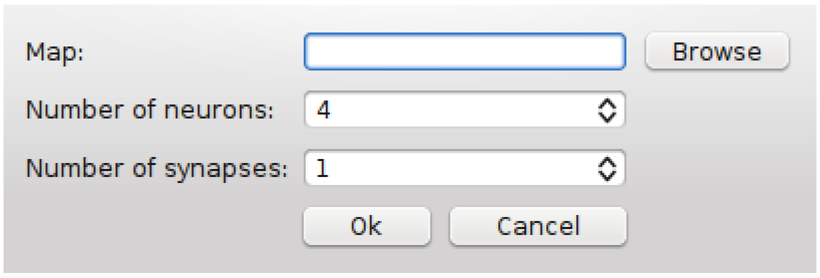
- "Save plots layout" - save the layout in which the plots are displayed and address from which they read the data.
- "Restore plots layout" - restores the layout of plots from the file.
- "Log data to a file" - logs data of all plots to a file as the html tables.
- "Load variables' map" - Loads the variables' map.

"Log data to a file"

When user clicks on this action he will be presented with dialog where he can input the name of the file where the logs in the form of html tables will be placed.

"Load variables' map"

The action with label "Load variables' map" shows a dialog.


Map:

Number of neurons: 4

Number of synapses: 1

Figure A.2: Variables' map loading menu of Ubiplot

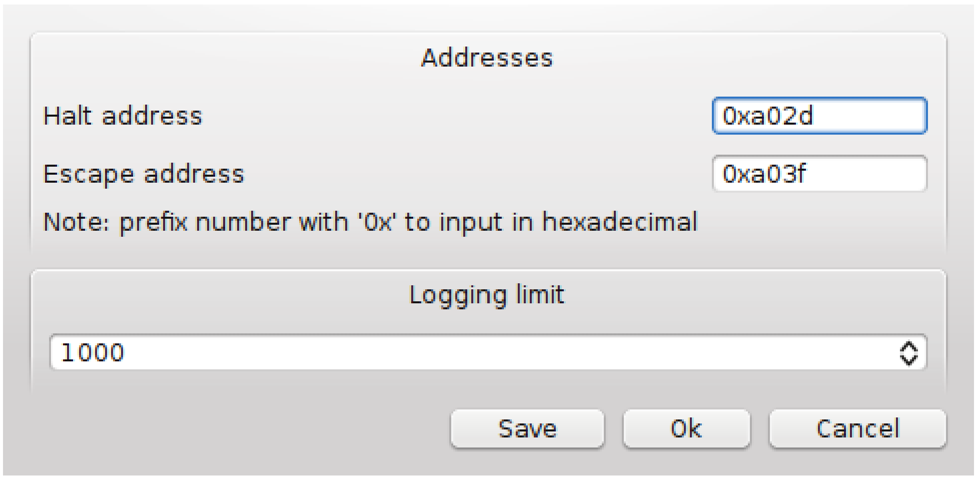
There user can point the desired map to load and the size of the network - number of neurons and synapses per each. If the name of the file has got two leveled extension the first one will be parsed in the search of the network size. For example file with name "map.4x3.yml" will be interpreted as map for network of 4 neurons with 3 synapses each. How to create a map is described in the section A.1.2.

View menu

Under "View" menu there are two actions "increase splitter size" and "decrease splitter size". When plots are created there are placed in the so called splitter in the main window. These actions will change the sizes of the plots. They are also accessible from the toolbar. A note, while creating plot it is also possible to not store the plot in the splitter but in the separate window. In that case the described actions will not affect the size of the plot.

Settings

The "Tools" menu holds one action "Settings" that shows dialog for configuring the plug-in.


Addresses

Halt address: 0xa02d

Escape address: 0xa03f

Note: prefix number with '0x' to input in hexadecimal

Logging limit: 1000

Figure A.3: Settings dialog of Ubiplot

There it is possible to change the address where the stop flag is stored and the address to which it is necessary to write in order to escape it. Besides that there is also possibility to change number of the execution steps for which data will be stored in the plots. The default value is 1000.

Besides the "Ok" button that applies current settings there is also "Save" button. Pushing it results in storing current settings, so they will be the same next time the plug-in is loaded.

Variables menu

After loading the variables' map the "Variables" menu becomes enabled. There user can pick type and for which units he wants to create a plot.

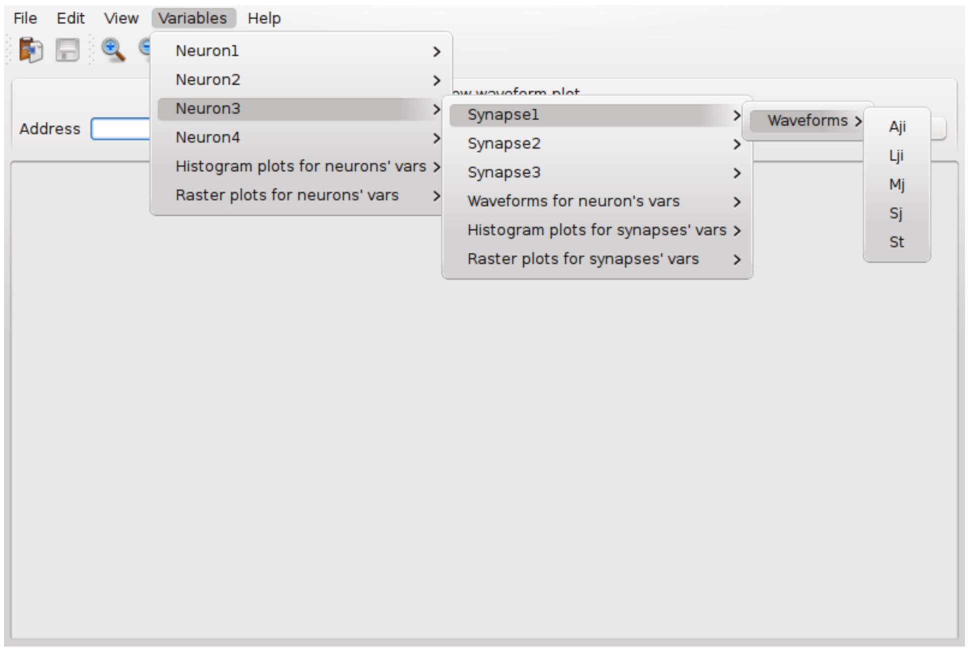
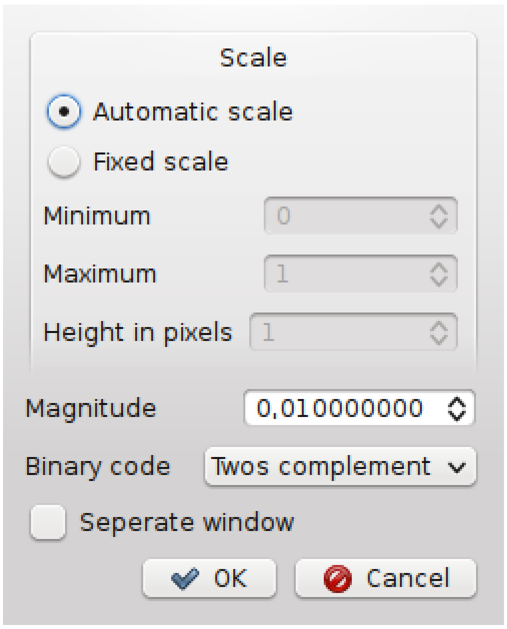

File
Edit
View
Variables
Help
Address
Neuron1
Neuron2
Neuron3
Neuron4
Histogram plots for neurons' vars
Raster plots for neurons' vars
Synapse1
Synapse2
Synapse3
Waveforms for neuron's vars
Histogram plots for synapses' vars
Raster plots for synapses' vars
Waveforms
Aji
Lji
Mj
Sj
St

Figure A.4: "Variables" menu of Ubiplot

Creating waveforms

Below the toolbar there is an input for creating waveform plot manually. There user can input the address and desired mask. After pushing apply button the start up dialog for waveforms will appear.


Scale

(●) Automatic scale

( ) Fixed scale

Minimum 0

Maximum 1

Height in pixels 1

Magnitude 0,010000000

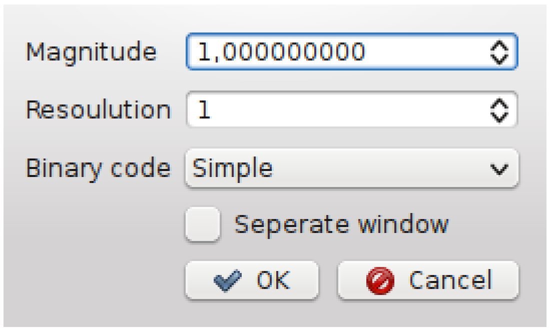
Binary code Twos complement

☐ Seperate window

Figure A.5: Start up dialog for waveforms.

There it is possible to fine tune the plot more. Choose whether it should auto scale, or to have fixed scale with inputed values (minimum, maximum and resolution). Choice of the used binary code is also available along with possibility to open the plot in the separate window. Plus the so called magnitude can be set which is the number by which the incoming values will be multiplied. The same dialog will be shown after clicking on waveform plot from "Variables" menu.

Creating histograms


Magnitude 1,000000000

Resoulution 1

Binary code Simple

Seperate window

Figure A.6: Start-up dialog for histograms.

Adding histograms is possible only from the "Variables" menu. The start-up dialog for them enables one to choose the binary system, the magnitude and to create separate window for this plot. These are the same options as for waveform plots. There is also field for specifying the desired resolution.

Creating raster plots

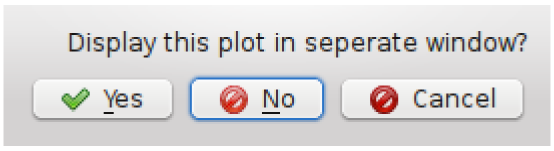

Display this plot in seperate window?

Figure A.7: Start-up dialog for raster plots.

Here the start-up dialog only asks if the plot should be placed in new window or in the main window.

Execution

In the toolbar user can find three buttons.

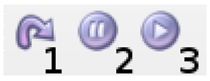


Figure A.8: Execution buttons.

As labeled in the Figure they are responsible for:

- 1 - executing one step.
- 2 - breaking.
- 3 - continuous run.

Context menu

The plots have got context menu available under the right click of a mouse. For histogram and raster plots menu has got only one entry which is "Close the plot". But for waveforms there are also "Connect points" and "Show list of data". Triggering the first of these entries will result in connecting the samples with each other. The second one will open a list with appended data to left of the plot. Then when right clicking on the list there is possibility to toggle the base system of the values between decimal and hex.

A.1.2 Variable's map

The map is needed by the plug-in to create the "Variables" menu. From this menu it is possible to choose to observe variable of given unit (neuron or synapse) or for all units (all neurons or all synapses of given neuron).

Depending on the indices of neuron and synapse, the address and mask for the request is different. The map holds expressions that are being evaluated by the plug-in to calculate the address, mask and other data needed for the request. The file where the map is placed is an yaml file. This format is both easy for software to parse and for humans to write and read. The map's file consists of three yml hashes whose keys are:

- define
- neuron
- synapse

They can be considered as a sections of the file and every one them is optional.

The value of the first hash is the list of strings. These strings are expressions that will be evaluated. They can be used to create JavaScript variables that can be used in the later expressions for not repeating the same code twice.

Next two hashes define the variables of neurons and synapses. Theirs keys are sequences of hashes. Each of the hash constitutes variable as follows:

- name - a name for identification
- mask - mask to use for masking out from the memory row
- delta - the difference between addresses of the each row if creating plot for more than one neuron/synapse
- binary code - binary code used
- magnitude - number to multiply the data by

Of course the above lacks the address. Since it is possible to access the variable by absolute address or by pointer two possible set of entries have to be added:

- address - direct address to variable

or:

- pointer - address to the pointer variable
- offset - how much to add to received address after dereferencing pointer

Only one of them should be used.

An example

Next an example map will be presented that defines two variables a neuron variable M_i and synapse variable L_{ji}.

```
neuron:
    - name: Mi
      pointer: SYNAPSES + 1
      offset: (N + 1)/2 - 1
      mask: "N%2 == 0 ? 0xFFFC0000 : 0x0000FFFC"
      delta: 1
      binary code: simple
synapse:
    - name: Lji
      pointer: S
      offset: NEURONS/2 + (N + 1)/2 - 1
      mask: "N%2 == 0 ? 0xFFFC0000 : 0x0000FFFC"
      delta: NEURONS
      binary code: simple
```

There are 4 JavaScript's variables that are set by the plug-in and can be used by the user - "NEURONS", "SYNAPSES", "N" and "S". The "NEURONS" and "SYNAPSE" are equal to the number of neurons and synapses of the network. While evaluating expression "N" and "S" are the indices of the neuron and synapse for which the request is being created.

Assuming user wants to plot L_{ji} for 1st synapse of 3rd neuron in the network of 4 neurons each having 3 synapses, the definition of L_{ji} should be read as follows:
Pointer to the variable is located at address that is equal to the index of the synapse (1 in this example).
After dereferencing it add $4/2 + (3 + 1)/2 - 1 = 3$ to get final address.
The mask will be 0x0000FFFC since the condition $N\%2 == 0$ for $N == 3$ is not met.
The difference between each address when plotting for more than one unit is equal to number of neurons in the network (4).
Treat data as coded in twos complement.
And multiply it by 0.01.

The quotation marks are needed when the string contains reserved keywords according to yaml specification (expression for mask in this case). If the entries "delta", "magnitude" or "binary code" will be omitted the sensible default values will be used (1, 1.0, "simple" respectively).

Above map can be less verbose if the "define" section will be used and the entries with default values will be omitted:

```
define:
    -
        "odd_even_neuron_idx = (N + 1)/2 - 1"
    -
        "sixteen_bits_mask = N%2 == 0 ? 0xFFFF0000 : 0x0000FFFF"
neuron:
    - name: Mi
      pointer: SYNAPSES + 1
      offset: odd_even_neuron_idx
      mask: sixteen_bits_mask
synapse:
    - name: Lji
      pointer: S
      offset: NEURONS/2 + odd_even_neuron_idx
      mask: sixteen_bits_mask
      delta: NEURONS
```

If user do not wants to use pointer he can input direct address in the map, making the entry for M_i even smaller:

```
neuron:
    - name: Mi
      address: 5 + (N + 1)/2 - 1
      mask: "N%2 == 0 ? 0xFFFC0000 : 0x0000FFFC"
```

Whole variable's map for code generated by SpiNDeK for Iglesias-Villa model is located in the appendix F. It is reflecting memory structure drowned in the figures 1.4 and 1.5.

The way the placement of the variables is defined by the map is very flexible. If in the future the memory structure will change writing new map should be a trivial task.

A.2 Building process

To build Ubiplot GNU g++ was tested but the code should be compile-able with any C++ compiler.

Ubimanager's plug-ins have to be linked with static library compiled from the part of the Ubimanager's source code. Within UbiManager's source code there is ubipluginfiles directory. In it, it is possible to execute make program for creating this library. The library is sufficient for most of the plug-ins. But for Ubiplot not all of the necessary symbols are included. That is why for development of this particular plug-in custom ubipluginfiles directory was created.

The building process consists of three steps:

- Compiling custom ubipluginfiles
- Compiling yaml-cpp library
- Compiling the plug-in itself

The first two are essential because Ubiplot have to be linked to these libraries.

The easiest way to build it would be to use git. *Git* is a source code management system. Although it is very easy to use on the Unix based systems but, by the time of writing this document, it is not on Windows. So first example with using git on Linux will be shown. Later without it on the Windows systems.

Building in GNU/Linux environment

The needed software on debian based Linux distribution can be obtained by typing these commands:

```
'sudo apt-get install qt4-dev build-essentials git '
```

Then:

```
`git clone git@asok:/ubiplot/ubiplot.git`
`git submodule init && git submodule update`
`cd ubipluginfiles && qmake && make`
`cd ../ubiplot/yaml-cpp && qmake && make`
`cd .. && qmake && make`
```

The first command fetches Ubiplot's code to the new local git repository. It will have as a submodule under the ubimanagertools directory that is a Ubimanager's repository. It is needed to build ubipluginfiles library. Next git commands initialize ubimanagertools with the newest code. Next commands build the ubipluginfiles and yaml-cpp library, and the plug-in itself.

Ubimanager's and Ubiplot's code is hosted on the gitorious.org website.

Building in windows environment

On windows the easiest approach would be to download the Qt SDK from the Qt's website. Its installer should install the Qt framework, the QtCreator which is an IDE and the minimalistic GNU for windows (MinGW) a toolchain for building C++ programs.

Next the source code would have to be obtained. Since it is cumbersome to use git on windows the best would be to get in some other way.

QtCreator can be used to compile the libraries and the plug-in. First the "ubipluginfiles/PluginLib.pro" would have to be opened with QtCreator and "Build project" should be chosen from the menu. This step would have to be repeated with "ubiplot/yaml-cpp/yaml-cpp.pro" and "ubiplot/ubiplot.pro" files. Thus created plug-in can be used on windows with the MinGW installed or with "QtScript4.dll" availale to the Ubimanager executable. Placing "QtScript4.dll" in the same directory as "Ubimanager.exe" will satisfy the last condition.

Final notes

It has to be noted that in order for plug-in to work with Ubimanager it has to be linked against the same version of it. Besides that both of them have to be compiled in debug or release mode (with or without debugging symbols) and Qt version used for compiling plug-in cannot be higher than the one used for Ubimanager.

A.3 Creating new kind of plot

To extend Ubiplot with new kind of plot one have to create three classes, for example: FooPlotWidget, FooPlot, FooPlotData. The inheritance should look like the existing one, that is FooPlotWidget should inherit from PlotWidget, FooPlot from Plot and FooPlotData from PlotData. As with already existing classes FooPlotData might have to be split in smaller classes that would abstract the used binary code or for other reason.

FooPlotWidget would have to implement the PlotWidget::toYML() method, FooPlot - Plot::paintEvent(), FooPlotData have to override PlotData::newData(), PlotData::toHtml() and PlotData::toYML().

The written code for this classes is placed in the appendix B. This example for simplicity sake do not log the data so toHtml() method returns empty string. Neither it is serializable so the methods toYML() have empty body.

That example will in every time step visualise which neurons fired. The constructor to FooPlotWidget accepts parameter *values* that indicates how many neurons there are to plot. Again for the sake of simplicity FooPlotData stores only spikes for the last step. And the task of FooPlot is to split available space in to rectangles that are red if spike occur or white otherwise.

Last thing to do would be to create a menu or toolbar entry in main window that would create a DataRequest object for spikes. Then call the FooPlotWidget constructor and setter for DataRequest on it. And lastly call PlotWindow::addPlotWidget() with newly created FooPlotWidget object as a parameter. Such method as an example is shown in the appendix B.4.

Appendix B

Example source code for a new kind of plot

B.1 FooPlotWidget class

```
#ifndef FOOPLOTWIDGET_H
#define FOOPLOTWIDGET_H
#include "plotwidget.h"
#include "fooplotdata.h"
#include "fooplot.h"
#include "math.h"

class FooPlotWidget: public PlotWidget{
    public:
        FooPlotWidget(int values,
                QWidget* parent): PlotWidget("Spikes visualisation", parent){
            FooPlotData* _plotData = new FooPlotData(values, this);
            plot = new FooPlot(sqrt(values), sqrt(values), _plotData, this);
            plotData = _plotData;
        }

        void toYML(YAML::Emitter&){ }
};
#endif
```

B.2 FooPlotData class

```
#ifndef FOOPLOTDATA_H
#define FOOPLOTDATA_H
#include "plot.h"
#include <QVector>
```

```
class FooPlotData: public PlotData{
    int values;
    public:
    QVector<bool> spikes;

    FooPlotData(int values, QWidget* parent): PlotData(parent){
        this->values = values;
    }

    void newData(unsigned short int* data){
        spikes.clear();
        for(int i = 0; i < values; ++i)
            spikes.append(data[i]);
    }


    void toYML(YAML::Emitter&){ }
    QString toHtml(){
        QString s;
        return s;
    }
};
#endif
```

B.3 FooPlot class

```
#ifndef FOOPLOT_H
#define FOOPLOT_H
#include "plot.h"
#include "fooplotdata.h"

class FooPlot: public Plot{
    int rows, cols;
    FooPlotData* plotData;

    public:
    FooPlot(int rows, int cols, FooPlotData* plotData, QWidget* parent):
        Plot(parent){
            this->rows = rows;
            this->cols = cols;
            this->plotData = plotData;
        }

    void paintEvent(QPaintEvent* /*evt*/){
        QVector<bool> spikes = plotData->spikes;
        if(spikes.isEmpty())
            return;
        QPainter painter(this);
        QBrush redBrush(Qt::red);
        QBrush whiteBrush(Qt::white);
        QPen pen(Qt::black);
        painter.setPen(pen);

        int width = QWidget::width()/rows;
        int height = QWidget::height()/cols;

        int i = 0;
        for(int r = 0; r < rows; ++r){
            for(int c = 0; c < cols; ++c){
                if(spikes.at(i++))
                    painter.setBrush(redBrush);
                else
                    painter.setBrush(whiteBrush);
                painter.drawRect(r*width,
                        c*height, (r+1)*width, (c+1)*height);
            }
        }
    }
};
#endif
```

B.4 PlotWindow class' slot for creating FooPlotWidget

```
void PlotWindow::spikeVisSlot(){
    if(!engine->isMapLoaded())
        return;
    QString si("Si");
    DataRequest* req = engine->createMultipleDataRequest(si, QPoint(1, 0));
    FooPlotWidget *widget = new FooPlotWidget(engine->getNeurons(), this);
    widget->setDataRequest(req);
    addPlotWidget(widget);
}
```

Appendix C

Initial values of the SpiNDeK's implementation of Iglesias-Villa model

Variable	Precision/Range	Initial Value (dec/hex)	Annotation
Vi(V_i)	16/-32768..32767	-7800/0x8800	Vi$_{initial}$ = Vrest
SumWeights($\sum w_{ji}$)	16/-32768..32767	-7800/0x8800	SumWeights$_{initial}$ = VRest
si(s_i)	1/0..1	X	1 bit for postsyn. spike
VRest1($V_{rest[1]}$)	16/-32768..32767	-7800/0xE188	scaled: -78mv = -7800
VRest2($V_{rest[2]}$)	16/-32768..32767	-7800/0xE188	scaled: -78mv = -7800
Theta1($\theta_{[1]}$)	16/-32768..32767	-4000/0xF060	scaled: -40mv = -4000
Theta2($\theta_{[2]}$)	16/-32768..32767	-4000/0xF060	scaled: -40mv = -4000
Dmem1	16/0..65535	61309/0xEF7D	stands for Kmem1: 14.99
Dmem2	16/0..65535	61309/0xEF7D	stands for Kmem2: 14.99
Mi(M_i)	14/0..16383	819/0x0333	Mi$_{initial}$: MMax/2
wji(w_{ji})	8/-128..127	-	no initial value needed
si(s_i)	1/0..1	X	1 bit for presyn. spike
Aji(A_{ji})	2/0..3	3/0x0003	Aji$_{initial}$ = 3
Lji(L_{ji})	14/0..16383	8191/0x1FFF	Lji$_{initial}$ = LMax/2
Mj(M_j)	14/0..16383	819/0x0333	Mji$_{initial}$ = MMax/2
LMax(L_{max})	14/0..16383	16383/0x3FFF	
MMax(M_{max})	14/0..16383	1638/0x0666	MMax = 1/10 of Lji
Pot1	8/-128..127	84/0x0054	scaled: 0,84mv = 84
Pot2	8/-128..127	-80/0xFFB0	scaled: -0,80mv = -80
Dsyn1	16/0..65535	63918/0xF9AE	stands for Ksyn1: 40
Dsyn2	16/0..65535	63918/0xF9AE	stands for Ksyn2: 40
Dact1	16/0..65535	65530/0xFFFA	stands for Kact1: 10922
Dact2	16/0..65535	65530/0xFFFA	stands for Kact2: 10922
Uno	1/-	1/0x0001	needed for increment

Appendix D

Assembler code of the SpiNDeK's implementation of Iglesias-Villa model

```
;4 Neurons, 3 Synapses
define size_x 4
define size_y 4
define synapses 2

.DATA

SYN-0="0cce0ccc,0ccc0ccc,7FFE7FFE,7FFE7FFE"
SYN-1="0cce0ccc,0ccc0ccc,7FFE7FFE,7FFE7FFE"
SYN-2="0cce0ccc,0ccc0ccc,7FFE7FFE,7FFE7FFE"

NEU-1="0ccc0cce,0cce0ccc"    ; Mi + Neuron Type + Si
NEU-2="E188E188,E188E188"    ; Vi
NEU-3="00000000,00000000"    ; Wji
NEU-4="1FFF1FFF,1FFF1FFF"    ; Tref + exponential

AMAX="00000003"
DACT1="0000FFFA"
DACT2="0000012C"
DBACK="0000FEB9"
DMEM1="0000EF7D"
DMEM2="0000EF7D"
DSYN1="0000F9AE"
DSYN2="0000F9AE"
LMAX="00003FFF"
MASK1="0000E000"
MASK2="0000C000"
MMAX="00000666"
POT1="00000054"
POT2="0000FFB0"
PROB="00001FFF"
SEED="A553A75A,A554A75A"
THETA1="0000F060"
THETA2="0000F060"
UNO="00000001"
VREST1="0000E188"
VREST2="0000E188"
MASC="00000003"
MASC1="000000FF"


.CODE

RST R0
SWAP R0
RST R0

RST R1
```

```
SWAP R1
RST R1

RST R2
SWAP R2
RST R2

RST R3
SWAP R3
RST R3

RST R4
SWAP R4
RST R4

RST R5
SWAP R5
RST R5

RST R6
SWAP R6
RST R6

RST R7
SWAP R7
RST R7

; ——————————————————————INIT SOME VARIABLES——————————————————————
LDALL R4,PROB
MOVA R4
SETMP SEED
RANDINI
RANDON
LOAD R1
RANDOFF
AND R1
MOVR R1
SWAP R1                              ;SR1 <-- activation probability
; ————————————————————————————————————————————————————————————————————

GOTO MAIN

; ————————————————————————————————————————————————————————————————————
; **************************** PROCEDURES BEGIN **************************

; —————————————————————————— NEURON LOAD ———————————————————————————
.NEURON_LOAD

SWAP   R6
LOAD   R6,NEU-2           ;SR6 <-- Vi
SWAP   R6

SWAP   R0
LOAD   R0,NEU-3           ;SR0 <-- SUM_WEIGHTS
SWAP   R0

; —————————————————————— Neuron Type + Si ——————————————————————

LOAD   R2,NEU-1       ;R2  <-- Mi + Neuron Type + Si
MOVA   R2
LDALL  R3,MASC
AND    R3
SWAP   R5
MOVR   R5                    ;SR5 <-- Neuron Type + Si
SWAP   R5

; —————————————————————————— Mi ——————————————————————————————
MOVA   R2
SHR
SHR
SWAP   R4
MOVR   R4                         ;SR4 <-- Mi
SWAP   R4

;————————————————————— Tref + exponential ————————————————————

LDALL  R3,MASK1       ;MASK1="0000E000"
SWAP   R5
MOVA   R5
SWAP   R5
SHR
SHR
FREEZENC
```

```
          LDALL  R3 ,MASK2    ;MASK2=" 0000 C000"
UNFREEZE

LOAD    R1 ,NEU-4              ;R1   <-- Tref + exponential
INV     R3                          ;MASK1 --> 1FFF  ;  MASK2 --> 3FFF
AND     R1
MOVR    R7                          ;R7   <-- 1FFF
SWAP    R7                   ;SR7 <-- exponential
MOVA    R1
AND     R3                               ;MASK1 = E000  ;  MASK2 = C000
MOVR    R7                          ;R7   <-- Tref

RET
 ; --------------------------------------------------------------------------------

 ; -------------------------------- MEMBRANE VALUE --------------------------------
.MEMBRANE_VALUE

RST   R1
RST   R2

SWAP  R5                            ;SR5 --> NEURON TYPE + Si

LDALL  R3 ,DMEM1               ;R3   <-- DECAY DONATOR 1
LDALL  R4 ,VREST1              ;R4   <-- Vres1

MOVA  R5
SHR
SHR                  ;IF  NEURON TYPE = TYPE_II  (CONDITIONAL LOAD)
FREEZENC
          LDALL  R3 ,DMEM2   ;R3   <-- DECAY DONATOR 2
          LDALL  R4 ,VREST2  ;R4   <-- Vres2
UNFREEZE

;------------------------ R2 <-- (1-Si(t))*(Vi(t)-Vres)*(Kmem) ------------------

MOVA  R5
SHR
FREEZEC              ;IF  (Si = 0)  THEN R2 <-- ((1)*(Vi(t)-Vres)*(Kmem)
          SWAP  R6                   ;SR6 <-- Vi
          MOVA  R6                   ;R0  <-- Vi
          SUB   R4                   ;R0  <-- Vi - Vres
          MOVR  R2                   ;R2  <--(Vi(t)-Vres)
          GOTO  DECAY                ;R2  =   (Vi(t)-Vres), R3 = DECAY DONATOR (1 or 2)
                                              ;R2  <--(Vi(t)-Vres) * (Kmem)
UNFREEZE

MOVA  R5
SHR
FREEZENC             ;IF  (Si = 1)  THEN R2 <-- ((0)*(Vi(t)-Vres)*(Kmem) = 0
     RST   R2                  ;R2   <-- ((0)*(Vi(t)-Vres)*(Kmem)
UNFREEZE

;---------- Vi <-- Vres + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS -----------

LDALL  R4 ,VREST1              ;R4   <-- Vres1

MOVA  R5
SHR
SHR
                               ;IF  NEURON TYPE = TYPE_II  (CONDITIONAL LOAD)
FREEZENC
          LDALL  R4 ,VREST2  ;R4   <-- Vres2
UNFREEZE

MOVA  R4                       ;R0   <-- Vres1 or Vres2
ADD   R2                       ;R0   <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)
MOVR  R2                       ;R2   <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem)

SWAP  R0                       ;R0   <-- SUM_WEIGHTS
ADD   R2                             ;R0   <-- (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS

MOVR  R6                             ;R6   <-- Vi = (Vres1 or Vres2) + (1-Si(t))*(Vi(t)-Vres)*(Kmem) + SUM_WEIGHTS
SWAP  R6                             ;SR6 <-- Vi

SWAP  R5                             ;SR5 <-- NEURON TYPE + Si

RST  R0                              ;SUM_WEIGHTS = 0
SWAP  R0                             ;SR0 <-- SUM_WEIGHTS

RET
 ; ------------------------------------------------------------------------------------
; ———————————————————————————— SYNAPSE LOAD ——————————————————————————

.SYNAPSE_LOAD
; ———————————————————————————————————————————————————————————————————————
; ———————————————————————————————— SP1 ————————————————————————————————
; ———————————————————————————————————————————————————————————————————————
; ———————————————————————— Mj + Synapse Type + Sj ——————————————————————
LDALL R1,MASC

SETC
SETMP SYN−0                                        ;LOOP INDEX
READMP 1
LOAD R2                                       ;R2  <−− Mj + Synapse Type + Sj


; —————————————————————————— Synapse Type + Sj ————————————————————————
MOVA R2
AND  R1
MOVR R6                          ;R6  <−− Synapse Type + Sj

; ———————————————————————————————— Mj ——————————————————————————————————
MOVA R2
SHR
SHR
MOVR R5                          ;R5  <−− Mj

; ———————————————————————————————————————————————————————————————————————
; ———————————————————————————————— SP2 ————————————————————————————————
; ———————————————————————————————————————————————————————————————————————

; ——————————————————————————————— Lji + Aji ———————————————————————————
LOAD R2                          ;R2  <−− Lji + Aji

; ———————————————————————————————— Aji ————————————————————————————————

SWAP R3
MOVA R2
AND  R1
RST  R1
MOVR R3
SWAP R3              ;SR3  <−− Aji

; ———————————————————————————————— Lji ————————————————————————————————
MOVA R2
SWAP R2
SHR
SHR
MOVR R2                          ;SR2  <−− Lji
SWAP R2
RET
; ———————————————————————————————————————————————————————————————————————

; ———————————————————————————— SYNAPTIC WEIGHT ————————————————————————
.SYNAPTIC_WEIGHT

RST R1
MOVA R6
SHR

FREEZENC                 ;IF (Sj = 1) THEN R0 <−− wji = Sj ∗ Aji ∗ P

        LDALL R4,POT1     ;R4  <−−  POT1
        MOVA R6
        SHR
        SHR
        FREEZENC
                LDALL R4,POT2
        UNFREEZE

;———————————————————————————————— Aji ∗ P ————————————————————————————————

        MOVFS R3
        MOVA R3
        SHR
        MOVR R3
        FREEZENC
                MOVA R1
                ADD R4
                MOVR R1
        UNFREEZE

        MOVA R3
        SHR
        MOVR R3
        FREEZENC
                MOVA R1
                ADD R4
                ADD R4
                MOVR R1
        UNFREEZE

        MOVFS R3
        MOVA R3
        SHR
        FREEZENC
                SHR
                FREEZENC
                        MOVA R1
                        ADD   R4
                        MOVR R1
                UNFREEZE
        UNFREEZE

UNFREEZE

SWAP R0
ADD R1                                  ; SR0 <-- wji = Sj * Aji * P
SWAP R0

RET
; ---------------------------------------------------------------------------

; ------------------------------ REAL_VALUE_VARIABLE ----------------------------
.REAL_VALUE_VARIABLE
;---------------Lji(t+1)  = Lji(t) * Kact + Si(t) * Mj + Sj(t) * Mi(t)--------
LDALL R3,DACT1                          ;R3  <-- DACT1
MOVA R6
SHR
SHR
FREEZENC
        LDALL R3,DACT2  ;R3  <-- DACT2
UNFREEZE

MOVFS R2                                        ;R2  <-- Lji

;------------------------------------ Lji(t) * Kact 1 or 2 -----------------------
GOTO DECAY          ;R2 <-- Lji(t) * Kact 1 or 2

SWAP R5
MOVA R5
SWAP R5
SHR                                             ;R5  <-- Si
FREEZENC
        MOVA R2
        ADD R5                              ;R0  <-- Mj
        MOVR R2                 ;R2  <-- (Lji(t) * Kact) + (Si(t) * Mj)
UNFREEZE
MOVA R6
SHR                                             ;R6  <-- Sj
FREEZENC
        MOVA R2
        SWAP R4
        SUB R4                              ;R0  <-- Mi
        SWAP R4
        MOVR R2                 ;R2  <-- (Lji(t) * Kact) + (Si(t) * Mj) + Sj(t) * Mi(t)
UNFREEZE
MOVTS R2                                    ;SR2 --> Lji
RET
; ------------------------------------------------------------------------------

; ---------------------------------- ACTIVATION_VARIABLE ----------------------------
.ACTIVATION_VARIABLE

LDALL R1,UNO
SWAP R3
MOVA R3
FREEZEZ            ;IF (Aji = 0) THEN
        LDALL R0,LMAX            ; (LMAX-LJI) < 0 ?
        SWAP R2
        SUB R2
        SHL
        FREEZENC                                ; --> YES
                MOVA R3
                ADD R1                          ; Aji + 1
                MOVR R3                         ; Aji --> SR3
                LDALL R0,AMAX
                SUB  R3                         ;  R0 <--   Amax  -  Aji
                SHL                                     ;  NEGATIVE?
                FREEZENC                        ;  Aji  -  R1  =  0
                        LDALL  R3,AMAX
                UNFREEZE
                LDALL  R0,LMAX     ;  LJI=LMAX/2
                SHR
                MOVR  R2
        UNFREEZE
                                                        ;  ELSE  IF  (LJI[J]  <  LMIN)  {
        MOVA  R2                                ;  LJI  --> ACC,  LMIN=0
        SHL                                             ;(LJI-LMIN(=0))  <  0  ?
        FREEZENC                                ;  (LJI-LMIN)  <  0  ?  YES
                MOVA  R3
                SUB  R1                         ;  AJI-1,  SR3  ACT.
                MOVR  R3                        ;  AJI  ->  R3
                LDALL  R0,LMAX     ;  LJI=LMAX/2
                SHR
                MOVR  R2
        UNFREEZE
UNFREEZE
MOVA  R3
FREEZENZ                                        ;IF  CONNECTION  IS  INACTIVE
        RST  R2
UNFREEZE
SWAP  R3
SWAP  R2
RET
; ------------------------------------------------------------------------------------------

; --------------------------------- MEMORY_OF_LAST_PRESYNAPTIC_SPIKE -----------------------
.MEMORY_OF_LAST_PRESYNAPTIC_SPIKE

; ----------------- Mj(t+1)  =  (Sj(t)  *  Mmax)  +  (1  -  Sj(t))  *  Mj(t)  *  Ksyn -----------------


;-------------------------- R2 <--  (1  -  Sj(t))  *  Mj(t)  *  Ksyn ------------------------------

LDALL  R3,DSYN1                         ;  R3 <--  Ksyn1

MOVA  R6                                        ;  R6 <--  Synapse  Type  +  Sj
SHR
SHR
FREEZENC
        LDALL  R3,DSYN2                 ;  R3 <--  Ksyn2
UNFREEZE

MOVA  R5                                        ;  R5 <--  Mj
MOVR  R2                                        ;  R2 <--  Mj
        GOTO  DECAY                ;  R2 <--  (1  -  Sj(t))  *  Mj  *  Ksyn1  or  Ksyn2


MOVA  R6                                        ;  R6 <--  Synapse  Type  +  Sj
SHR
FREEZENC        ;IF  Sj(t)  =  1  THEN  R2 <--  Mmax
        LDALL  R0,MMAX
        MOVR  R2                    ;  R2 <--  Mmax
UNFREEZE

MOVA  R2
MOVR  R5                                        ;  R5 <--  (Sj(t)  *  Mmax)  +  (1  -  Sj(t))  *  Mj(t)  *  Ksyn

RET
; --------------------------------------------------------------------------

; ------------------------------ SYNAPSE_SAVE ------------------------------
.SYNAPSE_SAVE

SETMP  SYN-0                            ;LOAD  LOOP  INDEX!
READMP  1               ;READMPX
;  *************************  1.  MJ+SI+TYPE  *******************************
MOVA  R6
SHR
SHL
MOVR  R6

MOVA  R5                        ;<--MJ
SHL
SHL
ADD  R6                         ;+TYPE+SJ
MOVR  R3                        ;composed  DATA

RST  R0
```

```
SHR
STNC R3  ;SAVE DATA


; ***************************** 2. LJI+AJI *********************************
SWAP R2
MOVA R2                    ;<--LJI
SWAP R2
SHL
SHL
SWAP R3
ADD R3                     ;+AJI
SWAP R3
MOVR R3                    ;composed DATA
RST R0
SHR
STNC R3                    ;SAVE DATA
RET
; ----------------------------------------------------------------------------

; ---------------------- MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE --------------
.MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE

LDALL R3,DSYN1                     ;TYPE=1
SWAP R5
MOVA R5
SHR
SHR                                                  ;--> TYPE
FREEZENC
        LDALL R3,DSYN2             ;TYPE=2
UNFREEZE
SWAP R4                                    ;R2=MI
MOVA R4
SWAP R4
MOVR R2
GOTO DECAY                                 ;R2=OPERAND, R3=DECAY DONATOR --> R2=RESULT DECAY
MOVA R5
SWAP R5
SHR                                                  ;-->SI
FREEZENC
        LDALL R0,MMAX
        MOVR R2                            ;OVERWRITE DECAY RESULT
UNFREEZE
MOVA R2
SWAP R4
MOVR R4                                    ;RES IN SR4
SWAP R4
RET
; ----------------------------------------------------------------------------

; --------------------------- SPIKE UPDATE -------------------------------
.SPIKE_UPDATE


LDALL R3,THETA1                 ;R3  <-- THETA1 = "0000F060"
SWAP R5                                    ;SR5 <-- Neuron Type + Si
MOVA R5
SHR
SHR
FREEZENC
        LDALL R3,THETA2         ;R3  <-- THETA2 = "0000F060"
UNFREEZE

MOVA R5
SHR
SHL
MOVR R5                                    ;R5  <-- Neuron Type + 0

SWAP R6                    ;SR6 <--  Vi
MOVA R6                    ;R0  <--  Vi
SUB  R3                                    ;R0  <--  Vi - (THETA1 or THETA2)
SWAP R6

FREEZENC
        MOVA R7                            ;R0  <-- refractary period
        SHL
        FREEZEC
                MOVA  R5
                LDALL R3,UNO
                ADD   R3
                MOVR  R5           ;R5  <-- Neuron Type + 1
                SET   R7                   ;R7  <-- activation of refractory time
        UNFREEZE
```

```
UNFREEZE
SWAP R5
RET
; ---------------------------------------------------------------------------------

; ------------------------------- BACKGROUND_ACTIVITY-------------------------------
.BACKGROUND_ACTIVITY

SWAP R7                     ; SR7 <-- exponential
MOVA R7
SWAP R7

MOVR R2                                  ; R2  <-- exponential
LDALL R3,DBACK              ; R3  <-- DBACK = "0000FEB9"
        GOTO DECAY                       ; R2  <-- DBACK * exponential

SWAP R1                                  ; R1  <-- activation probability
LDALL R4,PROB                    ; R4  <-- PROB = "00001FFF"
MOVA R4                                  ; R0  <-- PROB
SUB R2                      ; R0  <-- PROB - (DBACK * exponential)
RANDON
CLRC
SUB R1                                   ; (PROB - (DBACK * exponential)) - Activation probability
FREEZENC                    ;If ((PROB - (DBACK * exponential)) > Activation probability) then
        LOAD R1                          ; R1  <-- new activation probability
        RANDOFF
        MOVA R4                  ; R0  <-- PROB = "00001FFF"
        AND R1                   ; R0  <-- PROB = "00001FFF" AND new activation probability
        MOVR R1                  ;/ R1  <-- PROB = "00001FFF" AND new activation probability
        MOVA R4                          ; R0  <-- PROB = "00001FFF"
        MOVR R2                  ;/ R2  <-- PROB = "00001FFF"
        MOVA R7                          ; R0  <-- Tref
        SHL
        FREEZEC                          ; IF  ( C = 1 ) THEN Tref
                SWAP R5                  ; SR5 <-- Neuron Type + Si
                MOVA R5
                SHR
                SHL              ; SR5 <-- Neuron Type + Si = 0
                LDALL R3,UNO
                ADD R3           ; SR5 <-- Neuron Type + Si = 1
                MOVR R5
                SWAP R5
                SET R7                   ; R7  <-- activation of refractory time
        UNFREEZE
UNFREEZE
SWAP R1                                  ; SR1 <-- Activation probability
MOVA R2
SWAP R7
MOVR R7
SWAP R7                                  ; SR7 <-- Decay term
RET
; ---------------------------------------------------------------------------

; -------------------------------- REFRACTORY P ----------------------------
.REFRACTORY_P
MOVA R7
SHL                                      ; -1ms
MOVR R7
RET
; ---------------------------------------------------------------------------

; -------------------------------- NEURON SAVE -----------------------------
.NEURON_SAVE

SWAP R4                 ;R4  <-- Mi
SWAP R5                          ;R5  <-- Neuron Type + Si
SWAP R6

RST R3
MOVA R4
SHL
SHL
ADD R5
MOVR R3                 ;R3  <-- Mi + Neuron Type + Si

;--------------------------- INDIVIDUAL DATA STORE -------------------------

RST R0
SHR
STNC R3,NEU-1               ;SRAM  <-- Mi + Neuron Type + Si

RST R0
SHR
```

```
STNC R6,NEU−2              ;SRAM  <−− Vi

SWAP R0
CLRC
STNC R0,NEU−3              ;SRAM  <−− SUM_WEIGHTS
SWAP R0

LDALL R3,                  ;MASK1 = "0000E000"
MOVA R5
SWAP R5
SHR
SHR
FREEZENC
        LDALL R3,MASK2     ;MASK2 = "00008000"
UNFREEZE

MOVA R7                            ;ACC   <−− Tref
AND R3
SWAP R7                            ;R7    <−− exponential
OR R7
SWAP R7
CLRC
STNC R0,NEU−4              ;SRAM  <−− Tref + exponential
RET
; −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

;−−−−−−−−−−−−−−−−−−−−−−−−−−ENABLE SPIKES PROPAGATION−−−−−−−−−−−−−−−−−−−−−−−−−−
.SPIKES_ENABLE

SWAP R5                     ; ACC <== Spikes
MOVA R5
SWAP R5
SETC
SETMP SYN−0                 ; Point to Sj
READMP

RET
;−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

; −−−−−−−−−−−−−−−−−−−−−−−−−−−−−− EXPONENTIAL DECAY −−−−−−−−−−−−−−−−−−−−−−−−−−−−−
.DECAY
RST R1
MOVA R2
MOVR R4
SHL
FREEZENC
        RST R0
        SUB R2
        MOVR R2
UNFREEZE
LOOP 15
        MOVA R2
        SHL
        MOVR R2
        FREEZENC
                MOVA R1
                ADD R3
                MOVR R1
        UNFREEZE
        MOVA R3
        SHR
        MOVR R3
ENDL
MOVA R1
SHR
MOVR R1
MOVA R4
SHL
FREEZENC
        RST R0
        SUB R1
        MOVR R1
UNFREEZE
MOVA R1
MOVR R2
RST R1
RET
; −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
; *************************** PROCEDURES END *****************************

; *************************** MAIN PROGRAMME BEGIN ***********************
.MAIN
GOTO NEURON_LOAD
GOTO MEMBRANE_VALUE
LOOP synapses
        GOTO SYNAPSE_LOAD
        GOTO SYNAPTIC_WEIGHT
        GOTO REAL_VALUE_VARIABLE
        GOTO ACTIVATION_VARIABLE
        GOTO MEMORY_OF_LAST_PRESYNAPTIC_SPIKE
        GOTO SYNAPSE_SAVE
ENDL
GOTO MEMORY_OF_LAST_POSTSYNAPTIC_SPIKE
GOTO SPIKE_UPDATE
GOTO BACKGROUND_ACTIVITY
GOTO REFRACTORY_P
GOTO NEURON_SAVE
GOTO SPIKES_ENABLE
STOP
HALT
GOTO MAIN
; *************************** MAIN PROGRAMME END *************************
```

Appendix E

Implementation of the algorithm for filling map of HistogramPloData class

```
template<class T>
void HistogramPlotData<T>::fillMap(T *data, QMap<float, uint> *map){
    if(magnitude != 1.0)
        for(int i = 0; i < values; i++)
            data[i] = ROUND(data[i] * magnitude);
    //insert keys
    std::sort(data, data + values);
    T min = data[0];
    T max = data[values-1];
    map->insert(min, 1);
    float step;
    int i = 0;
    do{
        step = abs(max - min)/static_cast<float>(res);
        map->insert(min + (++i)*step, 0);
    }while(i < res && min != max);
    //insert values
    QList<float> keys = map->keys();
    int lastIdx = map->count() - 1;
    uint keyIdx = 0;
    //omit first sample since it was taken care of
    //while inserting keys
    uint dataIdx = 1;
    while(dataIdx < values){
        float key = keys.at(keyIdx);
        if(keyIdx == lastIdx){//the rest matches the last range
            uint & value = (*map)[key];
            value = values - dataIdx;
```

```
                if(value > maxQuantity)
                    maxQuantity = value;
                break;
            }
            if((data[dataIdx] > key && data[dataIdx] < key + step) ||
                    data[dataIdx] == key ){
                uint & value = (*map)[key];
                ++value;
                ++dataIdx;
                if(value > maxQuantity)
                    maxQuantity = value;
            }
            else
                ++keyIdx;
        }
    }
```

Appendix F

Ubiplot's variables' map for Iglesias-Villa model

```
define:
  -
    "odd_even_neuron_idx = (N + 1)/2 - 1"
  -
    "first_bit_mask = N%2 == 0 ? 0x00010000 : 0x00000001"
  -
    "second_bit_mask = N%2 == 0 ? 0x00020000 : 0x00000002"
  -
    "sixteen_bits_mask = N%2 == 0 ? 0xFFFF0000 : 0x0000FFFF"
neuron:
  - name: Si
    pointer: SYNAPSES + 1
    offset: odd_even_neuron_index
    mask: first_bit_mask
  - name: Nt
    pointer: SYNAPSES + 1
    offset: odd_even_neuron_index
    mask: second_bit_mask
  - name: Mi
    pointer: SYNAPSES + 1
    offset: odd_even_neuron_index
    mask: "N%2 == 0 ? 0xFFFC0000 : 0x0000FFFC"
  - name: Vi
    pointer: SYNAPSES + 2
    offset: odd_even_neuron_index
    mask: sixteen_bits_mask
    binary code: twos complement
    magnitude: 0.01
  - name: Swji
    pointer: SYNAPSES + 3
    offset: odd_even_neuron_index
```

```
        mask: sixteen_bits_mask
        binary code: twos complement
      − name: Exp(−lambda)
        pointer: SYNAPSES + 4
        offset: odd_even_neuron_index
        mask: "N%2 == 0 ? 0x1FFF0000 : 0x00001FFF"
      − name: Tref
        pointer: SYNAPSES + 4
        offset: odd_even_neuron_index
        mask: "N%2 == 0 ? 0xE0000000 : 0x0000E000"
synapse:
      − name: Sj
        pointer: S
        offset: odd_even_neuron_index
        mask: first_bit_mask
        delta: NEURONS
      − name: St
        pointer: S
        offset: odd_even_neuron_index
        mask: second_bit_mask
        delta: NEURONS
      − name: Mj
        pointer: S
        offset: odd_even_neuron_index
        mask: "N%2 == 0 ? 0xFFFC0000 : 0x0000FFFC"
        delta: NEURONS
      − name: Aji
        pointer: S
        offset: NEURONS/2 + odd_even_neuron_index
        mask: "N%2 == 0 ? 0x00030000 : 0x00000003"
        delta: NEURONS
      − name: Lji
        pointer: S
        offset: NEURONS/2 + odd_even_neuron_index
        mask: "N%2 == 0 ? 0xFFFC0000 : 0x0000FFFC"
        delta: NEURONS
```