ETH Zürich Universität Zürich

Institute of Neuroinformatics

An embedded neuromorphic computing platform for cognitive agents

Author:
Michel Frising

Supervisors:
Prof. Dr. Giacomo Indiveri
Dr. Yulia Sandamirskaya
Moritz Milde

June 9, 2016

Abstract

A autonomous mobile platform controlled by spiking neurons on the Reconfigurable On-line Learning Spiking (ROLLS) embedded neuromorphic processor is demonstrated. The neurons are stimulated with image events streamed from an embedded Dynamic Vision Sensor (eDVS) camera. The Parallella, a credit-card sized computer, serves as a host for the three devices.

Contents

1 Introduction

Humans and animals can effortlessly interact with dynamic environments - a feat that man-made robotic systems notoriously fail at. Perception is one of the bottlenecks limiting the realization of intelligent autonomous robots capable of navigating in unconstrained environments. Conventional image processing algorithms are known to be computationally expensive limiting their application in real-time scenarios[1]. They are also often specific for a given task, limiting their flexibility. Moreover state-of-the art image processing algorithms process frames containing a snapshot of the whole field of vision at a given moment[1]. These frames however contain a lot of redundant information within a series of consecutive frames and even within individual frames, stressing the limited memory of embedded systems unnecessarily[1]. Hardware implementations of models of biological neuronal architectures are a promising path to address these problems. Their output are often asynchronously timed spike trains (see for example the Dynamic Vision Sensor (DVS)[2] or the Reconfigurable On-line Learning Spiking (ROLLS) neuromorphic chip[3]). Neuromorphic circuits implemented with transistors operating in subthreshold also have the advantage of low power consumption, e.g. the Dynamic Vision Sensor (DVS)[2]. Moreover artificial neural networks are intrinsically capable of parallel processing like their biological counterparts. With the rise of the Internet Of Things, electronic devices become more and more interconnected and smarter such that low-power, fast and robust embedded computing platforms become ever more important.

The goal of this project was to demonstrate the feasibility of a neuromorphic computing platform processing asynchronous input events, e.g. input stream of an embedded Dynamic Vision Sensor (eDVS) camera, for controlling a mobile robotic platform. The interfaces to the different components and their interactions are defined with custom developed `C++` code, as explained further in section 2 on the following page. Two applications were developed with the help of this interfaces, a tool for ground truth acquisition and a demonstration of the robotic platform being controlled solely by the spiking output of the Reconfigurable On-line Learning Spiking (ROLLS) neuromorphic chip.

2 Interfaces and features

The setup included a host computer, the eDVS camera, the OmniRob and the Reconfigurable On-line Learning Spiking (ROLLS) neuromorphic processor interfaced by a host computer, either a Parallella board or a PC, and an optional keyboard for input. Figure 1 defines the interactions between the different components of the setup. The different components and their interfaces will be explained in detail in the next sections.

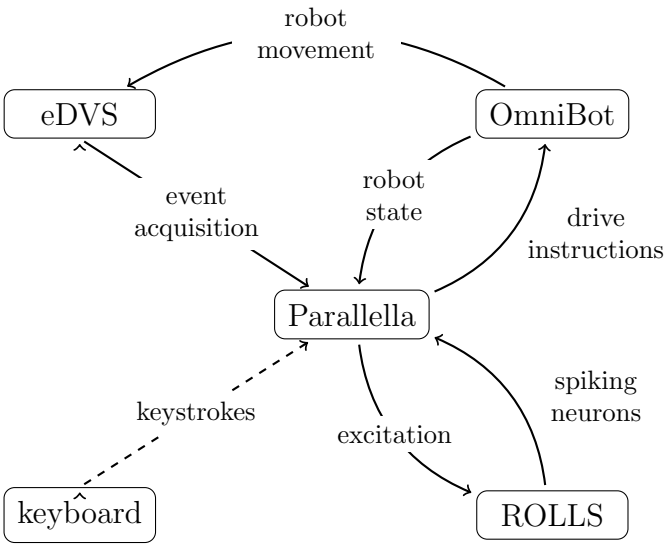

Figure 1. Overview of the different components of the experimental setup: a eDVS camera streams events to the Parallella minicomputer. The Parallella can stimulates neuron populations on the ROLLS and reads the addresses of the spiking neurons out and drives the mobile robotic platform, the OmniRob, accordingly. Alternatively the OmniBot can also be controlled with keystrokes with no feedback from the neuromorphic chip.

Two applications were realized in the scope of this work, first an application to acquire ground truth data for training of neural networks by controlling the robot movement by defined controls. For this application it is possible to drive the OmniRob with keystrokes from a host computer instead of the Parallella while simultaneously logging events from the eDVS and the drive state of the robot to gather ground truth data for learning experiments (see also section 3.1 on page 20). The second application shows the feasibility of the embedded loop in figure 1 and demonstrates an embedded neuromorphic computing platform. The abilities of this platform were demonstrated by implementing a simple application that was able to detect a blinking LED and orient the robot such that the LED is always in the center of the camera pane (see also section 3.2 on page 21).

2.1 Parallella

The Parallella from Adapteva [4] is a powerful credit-card sized mini-computer running a full fledged Linux operating system and serves as a host for all the applications showed in figure 1. The low power consumption (≈5 W[4]) and the high performance (tech specs from [4]) of the board make it an ideal platform for the intended setup:

- Zynq-Z7010 or Z7020 Dual-core ARM A9 CPU
- 16-core Epiphany Coprocessor
- 1GB RAM
- MicroSD Card Slot
- USB 2.0
- Up to 48 GPIO signal
- Gigabit Ethernet

- HDMI port
- Linux Operating System
- 54mm x 87mm form factor

As the Parallella runs a Linux operating system and has an Ethernet port and can also be fitted with a WiFi dongle, the board can be conveniently remote controlled with `ssh`:

```
# log in as normal user
ssh -X parallella@<IP Address>
# or to log in as super-user
ssh -X root@<IP Address>
# in both cases
Password: parallella
```

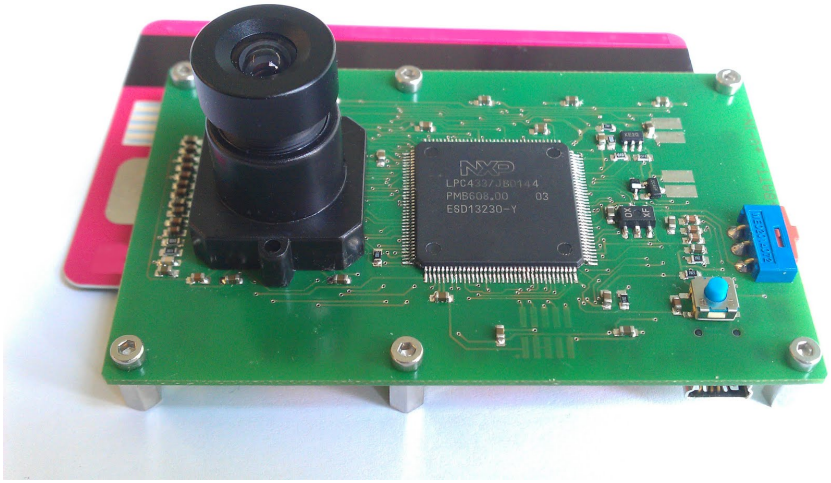

Figure 2. Photo of the low-power and small size eDVS camera from inilabs. Image from [5]

2.2 embedded Dynamic Vision Sensor (eDVS)

The eDVs is a spike-based silicon retina Dynamic Vision Sensor (DVS)[2, 5] that operates on a principle similar to the human retina resulting in a an asynchronous and sparse representation of visual stimulus discarding redundant image information. The pixels of the camera are continuous-time logarithmic photosensors that are capable of deciding individually when to digitize the analog vision signal[2]. The pixels have a high dynamic range (120 dB) as the pixels only react to logarithmic changes of the scene illumination and the output bandwidth is imposed by the dynamic parts of the scene[2]. One pixel typically has an area of $40\text{x}40\mu^2$ with 9.4 % fill-factor[2]. Figures 3 and 4 compare the DVS camera to a conventional camera. The digital signals emitted in form of address-events (AE) encode the relative change in reflectance with micro-second resolution timestamps providing fast asynchronous visual feedback[6]. One bit of the emitted AE also includes if the change in log intensity was positive or negative, also referred to as polarity. The dynamic vision sensor represents an attractive platform for low-latency dynamic vision in situations of uncontrolled illuminations and for embedded applications due to low data streams and processing requirements compared to frame-based systems with the same temporal resolution.[2]

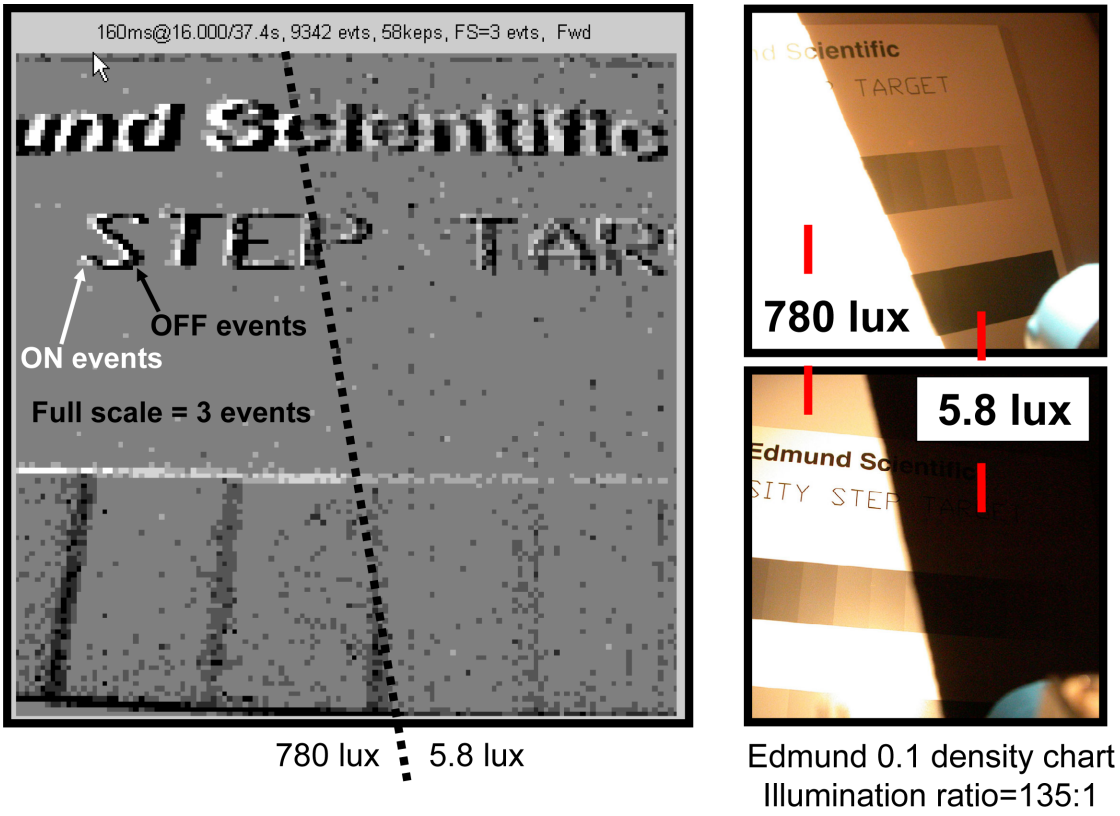

Figure 3. Demonstration of the dynamic range of the DVS. In the images on the right taken with a conventional camera half of the represented Edmund density chart is hidden by an artificially cast shadow, whereas the DVS camera has no problem with the widely varying scene illumination. From [7]

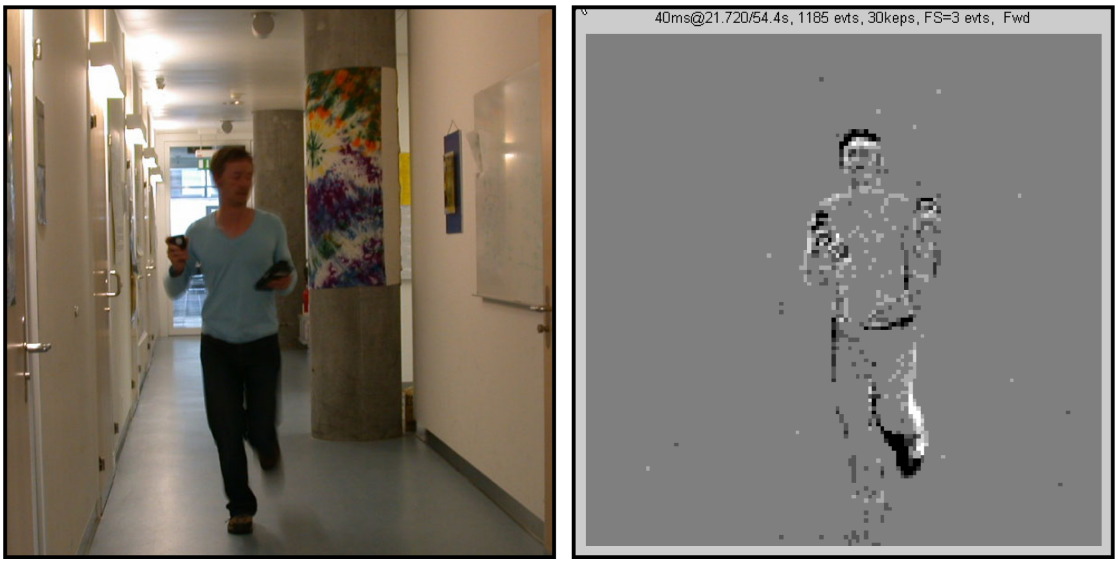

Figure 4. Demonstration of the dynamic capabilities of the DVS. On the left a still from a moving man and on the right the corresponding DVS image obtained by integrating the recorded events over a duration of $40\,\mathrm{ms}$. One can nicely see, that only the moving man triggers events, whereas the static background despite being rich in information, does not trigger additional events. From [7]

2.2.1 Interface

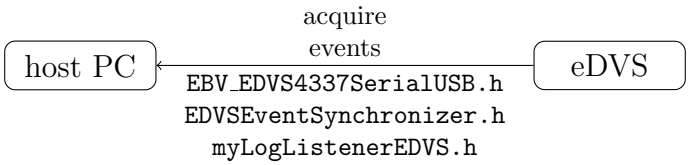

Figure 5. Header files defining the interfaces to the eDVS and the eDVS listeners

The used eDVS camera can be conveniently connected with an USB cable and interfaced as a serial device with the flags defined in listing 2.2.1, specified here [8], thanks to the embedded FTDI chip. The original code for this work was provided by Mr Julien Martel.

Listing 1. Configuration of the serial port according to [8]. The same configuration is used for the OmniBot

```
#include <linux/serial.h>
#include <termio.h>

int m_ttyFd = open(port_name, O_RDWR) /* Serial Port is opened
    as read-write on port port_name */
/*
The serial port is configured as non canonical mode, ie, raw
    mode.
*/
struct termios tty;
// get a copy of the current parameters
tcgetattr(m_ttyFd, &tty);
// set input mode flags
tty.c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP | INLCR |
    IGNCR | ICRNL | IXON);
// set output mode flags
tty.c_oflag &= ~OPOST;
// set local mode flags
tty.c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
// set control mode flags
tty.c_cflag &= ~(CSIZE | PARENB); /* No parity check */
tty.c_cflag |= CS8; /* 8 bits */
tty.c_cflag |= CLOCAL | CREAD; /* CLOCAL: ignore CD (carrier
    Detect) signal; CREAD: enable receiver */
tty.c_cflag |= CRTSCTS; /* use hardware flow control */
tty.c_cc[VMIN] = 0; /* minum number of charcters read before
    read returns */
tty.c_cc[VTIME] = 10; /* 1 second timeout */
```

The asynchronous nature of the AE-protocol is convenient for low-latency and low-redundancy data streaming, but poses challenges for traditional data acquisition. Typically sensors such as accelerometer or gyroscopes are read out at a fixed sampling frequency. We made heavy use of the Observer pattern[9] to handle the event-based nature of the DVS data transmission efficiently. The Unified Modeling Language (UML) class diagram in figure 6 on the following page explains the basic paradigm: the object `Subject` has a list of subscribed `Observer` that are notified with `notifyObservers()` upon change of the state of `Subject`. Observer can be added and removed to the list with `registerObserver(observer)` and `unregisterObserver(observer)`. The `Observer` object is a abstract base class with the purely virtual member `notify()`. `Observer` has to be subclassed and `notify()` has to be reimplemented to extend the observer with the desired behavior.

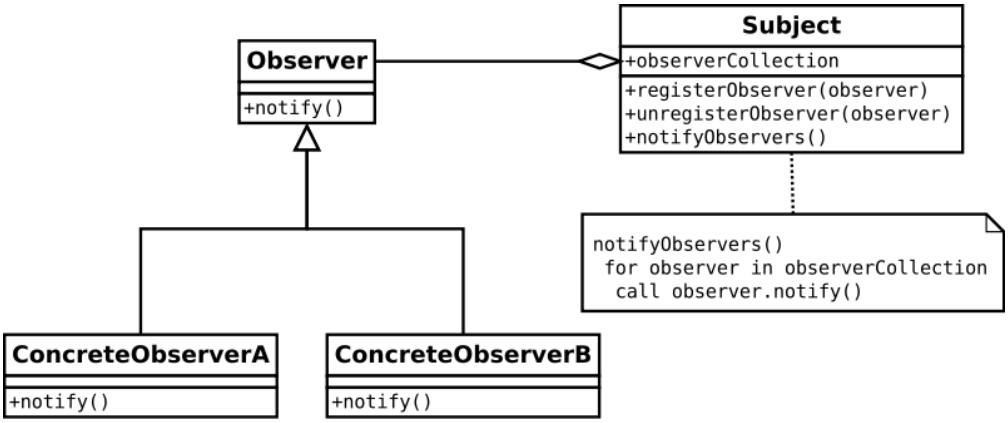

Figure 6. UML class diagram of the Observer Pattern. Image from [10]

Applying the observer pattern to the eDVS camera interface, results in the class structure in figure 8 on the next page. The denomination listener is preferred here over observer. The `Subject`, in this case the `EDVS4335SerialUSB` object has a dedicated thread that reads bytes from the serial port. The input is then parsed using the eDVS formats defined in [5]. Each time the chunk of bytes in the buffer is completely parsed, the subscribed listeners are notified with `warnEvent`. The eDVS also features an Inertial Measurement Unit (IMU) that is currently not used, but all the necessary interfaces are provided. Currently three listeners are provided, `myLogListenerEDVS`, `EDVSEventSynchronizer` and `DumbEDVS4337Listener`. The first one logs the eDVS events as comma-separated text (format: `x, y, timestamp, polarity`). The second one can perform a calibration to synchronize the timestamps provided by the eDVS with the clock of the computer (see also figure 7). The last listener simply displays all the received events in a window on the host computer screen like the black window on the right of figure 7.

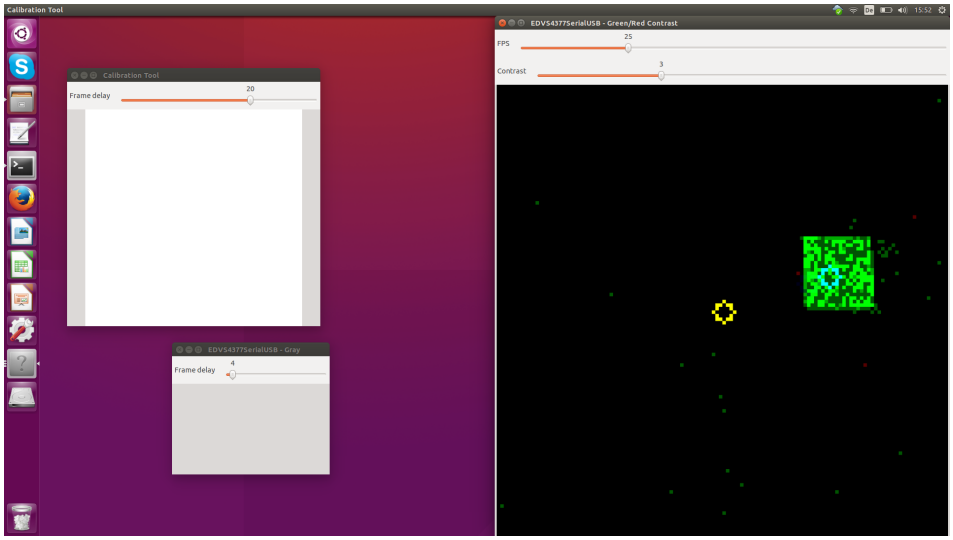

Figure 7. Image illustrating the synchronization process. When the synchronization process starts, a flashing window appears on the window. The DVS camera is oriented such that the flashing window is nicely visible in the event window (black window on the right). During the calibration the offset between the appearance of the programmed flashing window and the recording registered on the host computer is determined (typically 50 µs)

A typical usage example can be found in listing 2.2.1, showing how the interface to a eDVS camera connected over USB can be started and a listener can be registered and activated. More exhaustive information can be retrieved from the accompanying documentation [11] in the gitlab repository (`https://code.ini.uzh.ch/mmilde/omnibot-lib`).

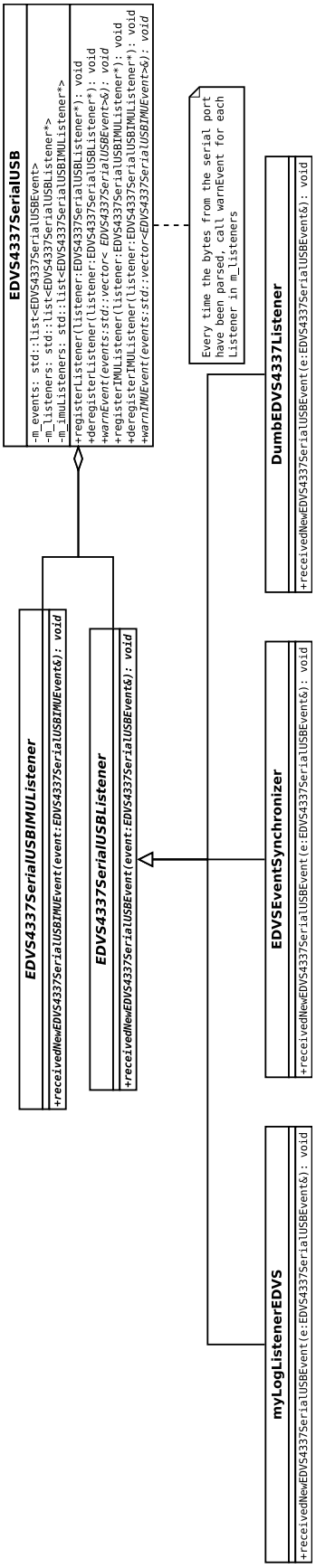

Figure 8. UML class diagram of the Observer Pattern applied on the eDVS interface.

Listing 2. Usage example for the eDVS interface with an event logger

```
#include "myLogListenerEDVS.h"
#include "EBV_EDVS4337SerialUSB.h"

// open a port to the eDVS camera
int dvsFileDescriptor = getUSBFileDescriptor(port_name_dvs,
   baudrate_dvs, "eDVS");
// initialize a eDVS device object with the valid file
   descriptor
// the Constructor takes care of setting up the serial port and
   starting the eDVS services
EDVS4337SerialUSB EDVSdevice(dvsFileDescriptor);
// Initialize an eDVS listener that logs to a file
myLogListenerEDVS EDVSlogger;
// register the listener with the openend camera
EDVSdevice.registerListener(&EDVSlogger);
//start listening
EDVSdevice.listen();
//stop EDVS
EDVSdevice.stopListening();
// and deregister the Listener
EDVSdevice.deregisterListener( &EDVSlogger );
```

2.3 Keyboard Input

The current version of the interface also allows to drive the robot with a keyboard connected to the host computer. At the moment only the keys in figure 9 are mapped to corresponding drive instructions, but the list can be extended easily if needed. Figure 9 also shows the intended robot movement when pressing the corresponding key. Releasing the key results in removing the according component from the movement, as rotational and translational movement can be superimposed independently for the OmniBot. The curved trajectories resulting from key combinations are not implemented yet.

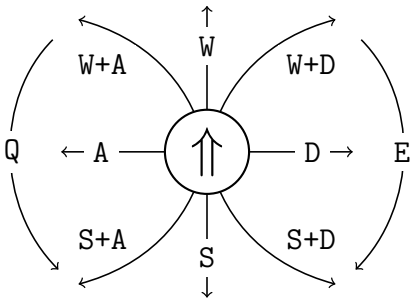

Figure 9. Mapped keystrokes and the resulting robot trajectories. The OmniBot is represented by the circle with the arrow, indicating the preferred direction (e.g. camera lens). The curved trajectories resulting from key combinations are not implemented yet.

2.3.1 Interface

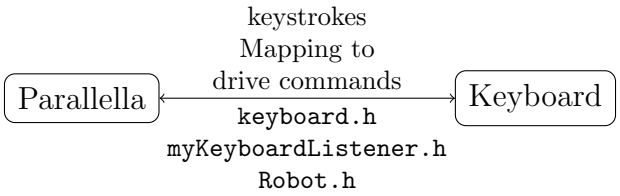

Figure 10. Header files defining the interfaces to the keyboard and the keyboard listeners

Detected key presses also arrive asynchronously and need to be processed as fast as possible, suggesting the use of the observer pattern as depicted in 11.

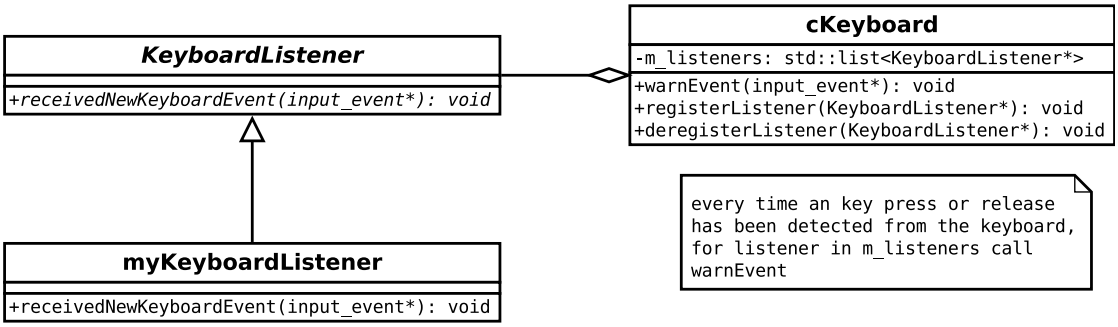

Figure 11. UML class diagram of the Observer Pattern applied on the keyboard interface.

The concrete implementation `myKeyboardListener` also has a pointer to the instance of the currently used robot, for handling the keystrokes. As shown in figure 12, pressed keys are converted to a bit set in the variable `int keysPressed`. Releasing the key reverts the bit at the corresponding position back to `0`. Non-supported keys are simply ignored.

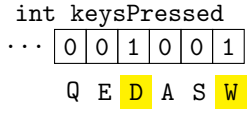

Figure 12. Example of how the registered key presses are stored in the variable `keysPressed`. Higher bits that are currently unused are not shown. When for example the keys D and W are pressed, the corresponding bits in the variable are set to `1` while all the other bits remain `0`.

2.4 OmniBot and PushBot

In the most recent version of the `omnibot-lib` two robots are supported: the OmniBot and the PushBot.

The OmniBot on the left in figure 13 on the following page is a mobile robotic platform with an omni-directional drive provided by NST at TU Munich. The robot has a diameter of approximately 30 cm and a large battery pack that allows it to operate up to 10 h without recharging (as specified by Mr Jörg Conradt in a conversation with Mr Moritz Milde). The planar top of the robot makes it an ideal substrate to mount all the modules from figure 1 on page 5 securely and make the experiment mobile. The OmniBot is fitted with a WiFi module, such that it can be accessed over telnet:

```
telnet omnibot1 56000
# the omnirob1 is registered in the INI Network
# type '??' to get a list of the currently supported features
```

We chose however to connect the robot over a serial port to the Parallella to make connection more robust and the small size, weight and power consumption of the Parallella allow it to be mounted on the robot as well.

The second supported robot is the PushBot, also provided by NST. The PushBot has a similar interface as the OmniRob and can also be easily accessed with the build-in WiFi module[8] over a socket application programming interface (API).

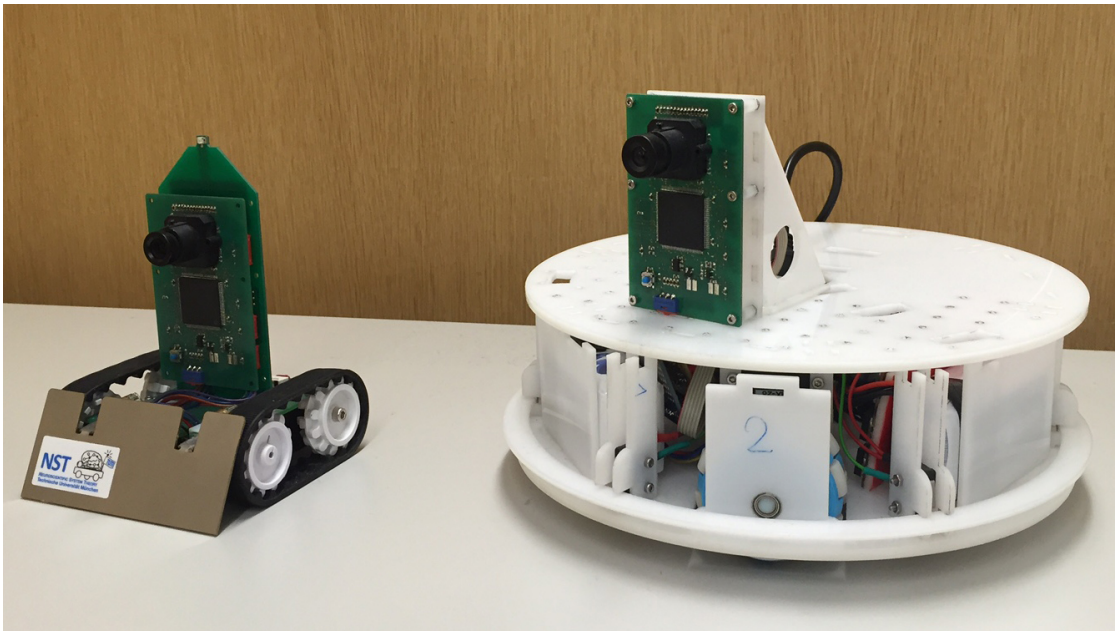

Figure 13. The two robotic platforms currently supported by `omnibot-lib`: on the left the PushBot and on the right the OmniBot (see also [12, 5]) for more information

2.4.1 Interface

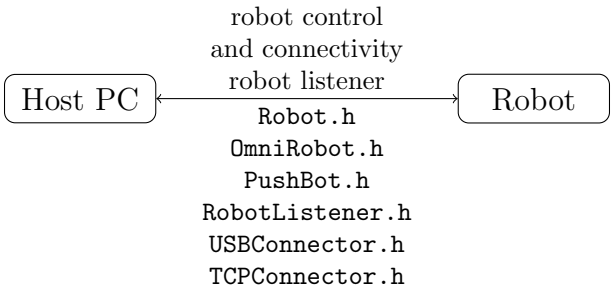

Figure 14. Header files defining the interfaces to the robots and the robot listeners

The Robot was controlled by sending the necessary instructions over the serial connection from the Parallella. The most important instructions are listed in table 1 on the following page:

Similarly to the DVS camera, different properties of the OmniBot are monitored asynchronously by the `RobotListener` object, such as the actual servo states, if and which bumper has been hit, making use again of the observer pattern as can be seen in figure 15 on page 15. These states are logged as formatted text or processed immediately, such as the bumper states, that trigger an emergency stop of the robot until the next drive command is send. At the moment the Robot is polled in an interval of 1 s, future updates of the OmniBot firmware will feature a broadcast function. However, logging the robot parameters is currently disabled as it is favored to directly log the mapped keyboard inputs (see also figure 9 on page 11 for a list of the mappings) instead of the polled robot parameters, amounting to the same result. Linux has

Table 1. Overview of the most important commands for the OmniBot. A complete list can be retrieved by sending on a console '??' to the OmniBot.

Instruction	Explanation
`!P<id><signal>`	Set the speed of servo `id` to `signal`. The signal can assume values from `-1024` to `1024` in integer steps, where `-1024`, `0` and `1024` correspond to the servos standing still. Maximum speed can be obtained with a signal around `900`. One unit of the signal corresponds roughly to 0.111 rpm. With a wheel radius of 2.54 cm this corresponds to a speed of $0.28\,\mathrm{cm\,s^{-1}}$ per unit signal resulting in a top speed of $2.8\,\mathrm{m\,s^{-1}}$.
`!PA<signal>`	sets all the servos to the same signal. Useful for turning on place.
`!T<enable><id><signal>`	Sets the torque of servo `id` to `signal`. enable is a flag assuming the values `1` and `0` to enable and disable the servo respectively.
`?SA`	Query the speed of every servo. Returns a string of the form `-SA <servo 0> <servo 1> <servo 2>`
`?B`	Query the battery voltage. Returns a string of the form `-B <voltage>`
`?Ib`	Query the value of the bumper. Returns a string of the form `-Ib: <bumper>`. `bumper` is an integer where the bits are **1** if the bumper is hit and `0` else. To determine which bumper has been hit a bitmask of the form `bumper & 0b10` (e.g, bumper 2) can be applied. The bit-shift of each bumper is indicated on the OmniBot in blue marker pen.

abstract interfaces to deal with physical devices, so called "file descriptors", such that the only time that the physical connection, i.e. serial connection or wireless, has to be taken into account is when opening the file descriptor.

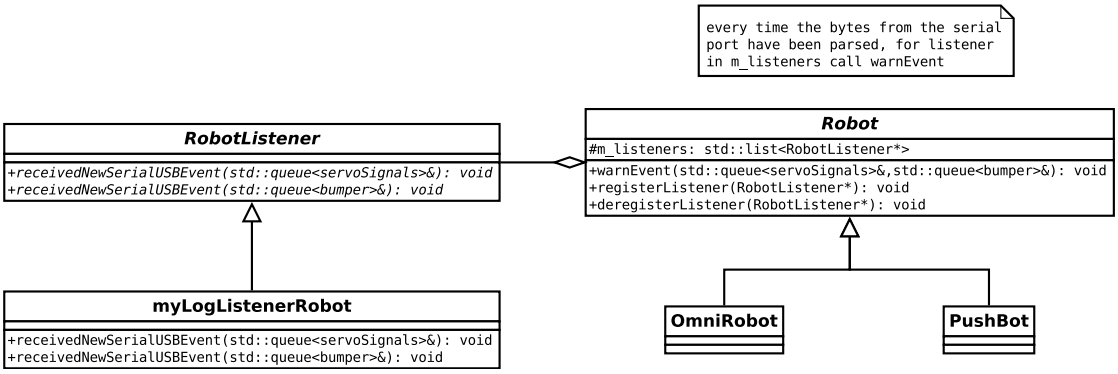

Figure 15. UML class diagram of the Observer Pattern applied on the robot interface.

Additionally a custom drive function was developed according to the ideas from [13] to provide high-level commands to steer the robot. The three omni-directional wheels of the OmniBot allow it to move in every direction immediately by turning on the place, unlike a conventional car for example, making it holonomic. The idea behind motor control of this robot is projecting the two-dimensional vector giving the speed and direction of the wanted movement on each wheel direction, as depicted in figure 16.

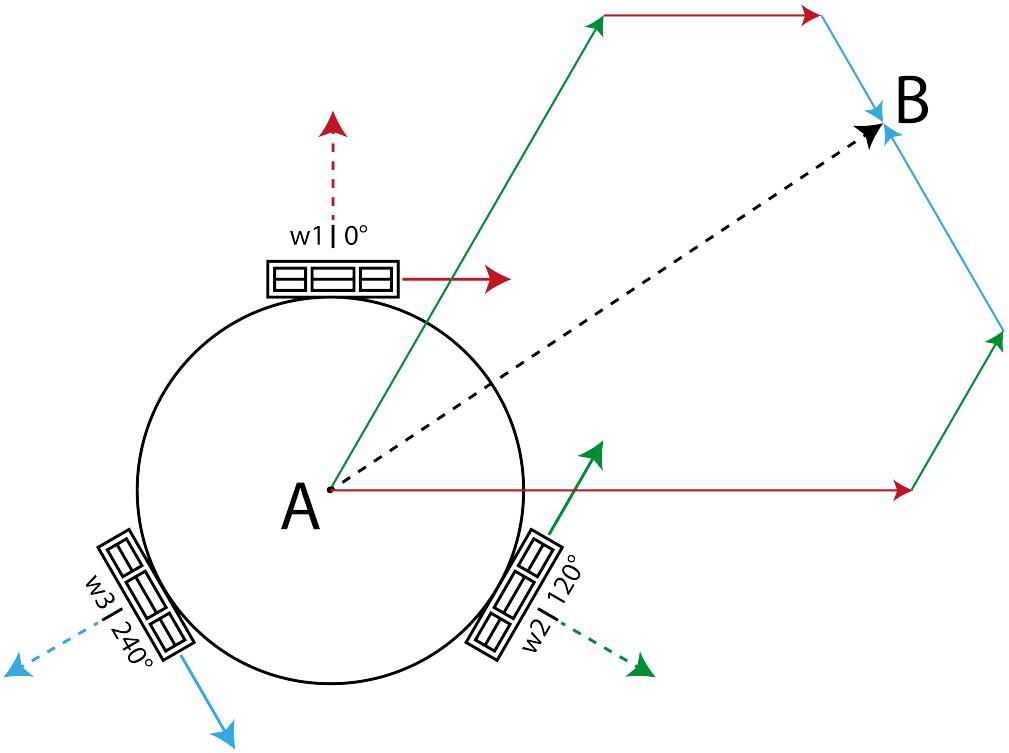

Figure 16. Schematic of the circular chassis of the OmniBot. The wheels are arranged tangentially on the periphery of the chassis, with a spacing of 120° between them. The solid arrows designate the wheel directions that can be actively controlled by the servos. The dotted arrows designate the directions that the wheels can passively turn. A vector (AB) denoting the trajectory of the robot from A to B is decomposed in the active directions of the wheel space. As the robot has 6 degrees of freedom and the trajectory only gives 2 constraints (as the robot is centrally symmetric), more than one decomposition of (AB) is possible. Image adapted from [14]

The drive function has the following signature:

```
void drive(double angular_vel, bool CCW, double linear_vel,
    double direction)
```

`angular_vel` and `linear_vel` are the desired angular and linear velocity respectively and are normalized with respect to their maximum values and have an allowed range of `0` to `100`. The speed coordinates are given in polar coordinates, hence `linear_vel` and `direction` (in degrees) must be provided instead of v_x and v_y. The flag `CCW` indicates if a angular movement should be counterclockwise (`CCW=true`) or clockwise (`CCW=false`). Listing 3 gives a simple example of typical usage of the OmniBot interface. More details about the implementation can be found in the accompanying documentation on gitlab.

Listing 3. Robot Interface usage example

```
#include "OmniRobot.h"
#include "PushBot.h"
#include "USBConnector.h"
#include "TCPConnector.h"
#include "myLogListenerRobot.h"

int robotFileDescriptor = getUSBFileDescriptor(port_name_robot,
    baudrate_robot, "OmniRob");
OmniRobot omniRobot(robotFileDescriptor);
omniRobot.drive(30,true,0,0); // drive in a circle
omniRobot.drive(0,true,30,0); // drive straight ahead

// register a logging listener to the robot
myLogListenerRobot myRobotLogger;
omniRobot.registerListener( &myRobotLogger );
omniRobot.listen();
/* do some stuff here */
// stop robot
omniRobot.stopListening();
//and deregister Listener
omniRobot.deregisterListener( &myRobotLogger );
```

2.5 Reconfigurable On-line Learning Spiking (ROLLS) Neuromorphic Processor

The Reconfigurable On-line Learning Spiking (ROLLS) neuromorphic processor developed at INI is a highly integrated (on-chip area = 51.4 mm) and low-power (4 mW for typical experiments) VLSI device. Different types of synapses and neurons on the device emulate the biologically plausible behavior of real neurons and synapses. 256 neurons are shared by 256×256 learning synapses for long-term plasticity and 256×256 programmable synapses as seen in figure 17 on the following page. Additionally 256×2 virtual synapses for modeling excitatory and inhibitory synapses are integrated. The device is aimed at exploring computational science models for brain-inspired circuits. Integrated bi-stable spike-based plasticity mechanisms also provide on-line learning abilities.[3].

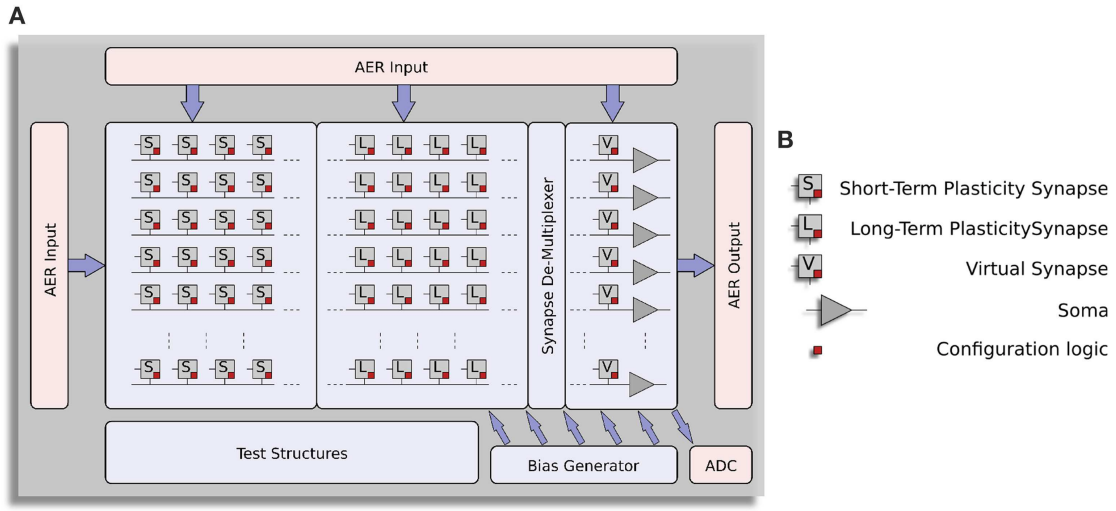

Figure 17. Block diagram of the layout of the ROLLS processor. The first block from left to right represents the programmable synapses, the second block the learning synapses and the last block the virtual synapses and the soma of the neurons. Around the neurons and the synapses peripheral circuits providing the AER logic, an analog-to-digital converter and a temperature-compensated programmable bias generator to set the bias currents controlling the synapse and neuron properties.

2.6 Short-Term Plasticity Synaptic Array

In this work only the Short-Term Plasticity Synaptic Array was used for experiments. Mr. Ning Qiao suggested that it is always a good idea to turn unused features off by setting the respective currents to the lowest possible value to avoid any unwanted cross-talk between components. As mentioned in section 2.5 on the previous page, the programmable on-chip bias generator can be used to set the biases of the different components on the chip, e.g. the integrate and fire (IF) neurons or the Synaptic Array in figure 17 and tune the behavior of the different components. 3 provides a list of the controllable biases of the Short-Term Plasticity Synaptic Array, 2 the controllable biases for the IF neurons.

2.6.1 Interface

The interface to the ROLLS neuromorphic chip is provided by `aerctl.h` developed by Mr Jonathan Binas at INI, along with a webtool webAEX to visualize the spiking output of the chip. Documentation for the ROLLS chip can be mainly found in [3] and a accompanying manual written by Mr Ning Qiao.

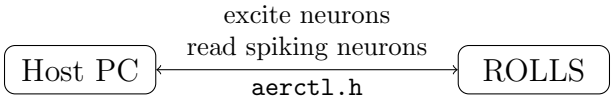

Figure 18. Header files defining the interface to the ROLLS

The ROLLS chip can be interfaced with the functions defined in `aerctl.h`, allowing to set connectivities between neurons, stimulating synapses and retrieving the addresses of the spiking neurons from the ROLLS that can then be used to control the robot(see also section 3 on page 20). However connectivities are not set in with the functions provided by `aerctl.h`, except for the stimulating input, but in a separate java-script file that can be loaded in the webAEX application developed by Mr Jonathan Binas that allows at the same time to display the retrieved spiking neurons with temporal and spatial resolution.

Table 2. List of the controllable biases for the Integrate and Fire (IF) neuron array

Bias Name	Effect
IF_RST_N	Controls the reset threshold current
IF_BUF_P	Buffer for Oscilloscope
IF_ATHR_N	Controls the adaption threshold current. The higher the more synaptic current is supplied
IF_RFR1_N	Controls the current that is used to discharge the capacitors controlling the refractory period of the neuron when RFR1 latch is storing 1. Cannot be <1 nA
IF_RFR2_N	Controls the current that is used to discharge the capacitors controlling the refractory period of the neuron when RFR1 latch is storing 1. Cannot be <1 nA
IF_AHW_P	Controls the adaption weight current
IF_AHTAU_N	Control the adaption time constant
IF_DC_P	Controls the value of additional static current that is injected in all neurons
IF_TAU2_N	Controls the time constant of the neuron when TAU2 latch is storing 1. The higher this value is the higher the time constant[a]
IF_TAU1_N	Controls the time constant of the neuron when TAU2 latch is storing 1. The higher this value is the higher the time constant
IF_NMDA_N	Controls the sensitivity of the neuron. The higher this current is, the easier the neuron will fire.
IF_CASC_N	Controls the cascode current
IF_THR_N	Controls the threshold current. The higher this value is the higher the synaptic current.

[a] If $I_w \gg I_\tau$, the static current through the diff-pair integrator (DPI) is given by [15]:

$$I_{\text{static}} = \frac{I_{\text{thr}} I_{\text{w}}}{I_\tau}$$

Table 3. List of the controllable biases for the Non-plastic synaptic Array (NPA). Adapted from [16]

Bias Name	Effect
NPA_PWKL_P	Controls the pulse-width of the synaptic current.
NPA_WEIGHT_STD_N	Controls the depression magnitude of the excitatory synaptic current after an input spike. The larger it is, the more depressed the synapse is after an input spike.
NPA_WEIGHT_EXC_P	Controls the magnitude of the weight independent component of the excitatory synaptic current. This component is always generated in response to an input spike when the synapse is excitatory irrespective of the 2-bit weight in the synapse.
NPA_WEIGHT_EXC0_P	Controls the magnitude of the excitatory current component which is injected into the excitatory DPI when the weight_0 latch is storing 1.
NPA_WEIGHT_EXC1_P	Controls the magnitude of the excitatory current component which is injected into the excitatory DPI when the weight_1 latch is storing 1.
NPDPIE_THR_P	Controls the threshold current of the excitatory DPI
NPDPIE_TAU_P	Controls the time constant of the excitatory DPI
NPA_WEIGHT_INH_N	Controls the magnitude of the weight independent component of the inhibitory synaptic current. This component is always generated in response to an input spike when the synapse is inhibitory irrespective of the 2-bit weight in the synapse.
NPA_WEIGHT_INH0_N	Controls the magnitude of the inhibitory current component which is injected into the inhibitory DPI when the weight_0 latch is storing 1.
NPA_WEIGHT_INH1_N	Controls the magnitude of the inhibitory current component which is injected into the inhibitory DPI when the weight_1 latch is storing 1.
NPAPDPII_TAU_P	Controls the threshold current of the inhibitory DPI
NPDPII_THR_P	Controls the time constant of the inhibitory DPI

3 Applications

A robust tool for ground data acquisition is necessary to generate reproducible real-world data for simulations to validate experiments or explore pathways for future experiments.

The second application demonstrates the feasibility of an embedded neuromorphic computing platform for cognitive agents by combining the DVS camera, a mobile robotic platform and the ROLLS neuromorphic chip.

3.1 Command line tool for ground truth data acquisition

The command line tool allows to log DVS events and the corresponding robot controls as formatted text to provide ground truth data of the mobile system for training and simulations. When all the components are up and running the program located can be started by typing in the command line:

```
// ./robotloop is located in
// /omnibot-lib/USB_robot/test/build
./robotloop
```

This assumes that the eDVS camera is connected on `/dev/ttyUSB0` and the OmniRob on `/dev/ttyUSB1`. The serial ports are opened with the default values 6000000 and 2000000 respectively. If no synchronization between the DVS camera and the host computer clock is performed, the robot can be controlled right away and the offset is set to the standard value of 50 µs. When synchronization is required, a window flashing at a fixed frequency pops up. After aligning the eDVS with the flashing window, synchronization is performed by determining the time offset between the recorded events and when the window has been flashed on the computer. The determined offset can then be used to eliminated the accumulated lag from the eDVS events and the robot events. `./robotloop` can also be called with number of optional arguments. When no arguments are supplied, the default values indicated in lsiting 4 are used: these values can be adjusted in `main.cpp`.

Listing 4. Usage examples of the command line tool for ground truth data acquisition

```
/* connecting a robot with a certain connectivity */
./robotloop [connectivity] [robot_type]
//default values:
// connectivity = serial
// robot_type = OMNI_BOT


/* connecting the push_pot */
./robotloop tcp push [host] [port]
//default values:
// robot_type = OMNI_BOT
// host = 10.162.177.186
// port = 56000

/* connecting a robot and a camera on a serial port with
   specified baudrates */
./robotloop [connectivity] [robot_type] [camera_port] [
   robot_port] [camera_baudrate] [robot_baudrate]
// the connectivity argument has to be provided, but has no
   effect
//default values:
// camera_port     = /dev/ttyUSB0
// robot_port      = /dev/ttyUSB1
// camera_baudrate = 6000000
// robot_baudrate  = 2000000
```

The ground truth data are stored in `/omnibot-lib/USB_robot/test/build` in a directory with the date as name (Format `YYYYMMDD`). Inside the directory the eDVS events are stored as formatted text in a file (log file name: `EDVS_HH_MM`, individual events: `x, y, timestamp, polarity`) as well as the robot drive instructions parsed from the keyboard input (log file name: keyboard_HH_MM, individual events: timestamp, `keysPressed`. `keysPressed` needs to be processed as a binary number and the keystrokes can be reconstructed with the help of figure 12 on page 12). In a future release of the OmniBot firmware from NST a broadcast function is planned, transmitting diagnostics for the OmniBot. These could then be monitored as well by adapting the `RobotListener` accordingly.

3.2 LED tracker

The second application developed was a simple demonstration how a robot controlled by the spiking neuron output of the ROLLS can follow the position of an LED (f_{LED}=60 Hz) waved in front of the eDVS camera. This application was completely embedded on the Parallella and an external computer was used only to display diagnostics. As seen in figure 19, the idea was to separate the upper part of the DVS retinal space in three regions, effectively reducing the horizontal resolution from 128 to 3: left, center, right. The lower part is going to be used for obstacle avoidance in future works. On the ROLLS chip three neuron populations in the range 0 to 150 were defined and grouped by 50 (without any connection among each other) corresponding to the DVS pane regions l(eft), c(enter), r(ight). Ideally the flashing LED generates much more events than anything around in the corresponding region of the DVS pane. The vents are then sorted to the different regions and the corresponding neuron population is excited. If the biases are well chosen, only the neuron population corresponding to the region where most spiking events were detected, i.e. where the LED is placed, is spiking, rejecting noise to some extent. The spiking neurons in turn are used to estimate the center of mass (COM) of the object, i.e. the position of the LED if no noise or other objects is in the background. The robot was controlled turn on place to follow the LED such that the LED always stays in the center of the DVS pane. Rotating the camera however generates a lot of spiking events, such that a noisy background degrades the performance of the simple LED tracker. By using a blank paper background this problem could be eliminated.

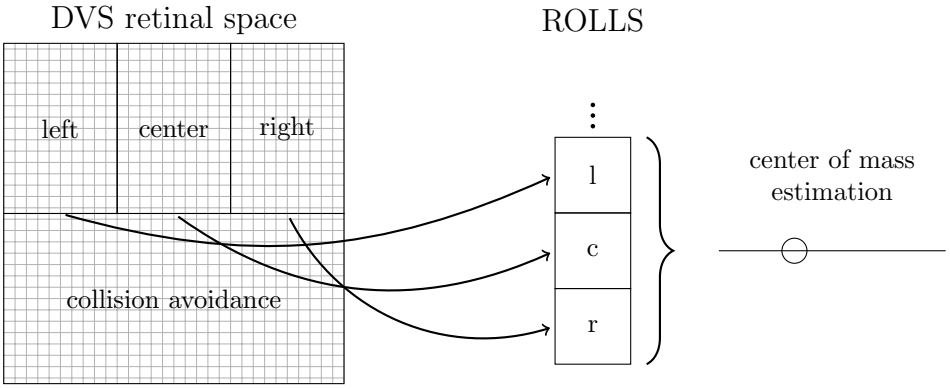

Figure 19. Schematic overview of the LED tracker setup. The top half of the eDVS is divided into three regions (left, center, right). On the ROLLS chip the neurons from 0 to 150 are grouped by 50 without any connections among each other (l(eft), c(enter), r(ight)). When an event is detected in one of the three regions, the corresponding group of neurons is excited. When the biases are well chosen, only the neuron population corresponding to the region where most spiking events were detected, i.e. where the LED is placed, is spiking, rejecting noise to some extent. The spiking neurons in turn are used to estimate the center of mass of the object, i.e. the position of the LED if no noise or other objects is in the background.

After dropping 90 % of the events coming from the DVS, one can nicely see in the right part of figure 21 on page 23 the flashing LED, while most of the background or other unwanted events are suppressed. In the left part of figure 21 on page 23 shows the corresponding spiking

neuron populations as defined in figure 19 on the preceding page. Indeed only the populations corresponding to the DVS pane region containing the bulk of the events spike. The spikes rapidly decay when the stimulation is stopped or the LED moves to another region. Only the three biases framed in figure 20 of the NPA block needed to be tuned to achieve this behavior. The exact biases and and a guide on how to use the ROLLS in combination with the Parallella can be found in section A in the appendix.

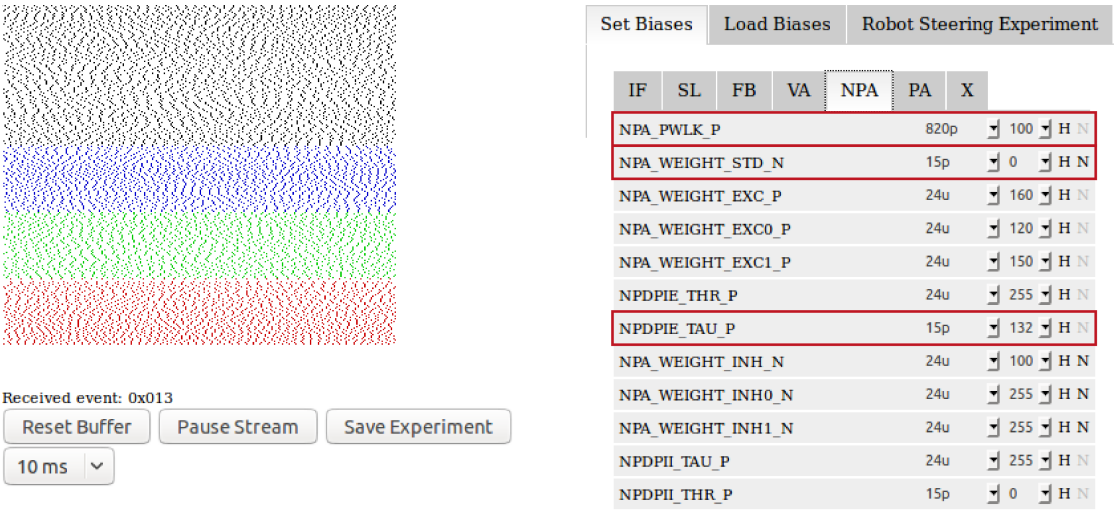

Figure 20. Screenshot of the webAEX tool with the tab allowing to change the NPA biases opened. The red frames indicate the three biases that were tuned to achieve the spiking behavior shown in figure 21 and 22. In this figure all the neurons are spiking even with no stimulation because the biases are ill tuned.

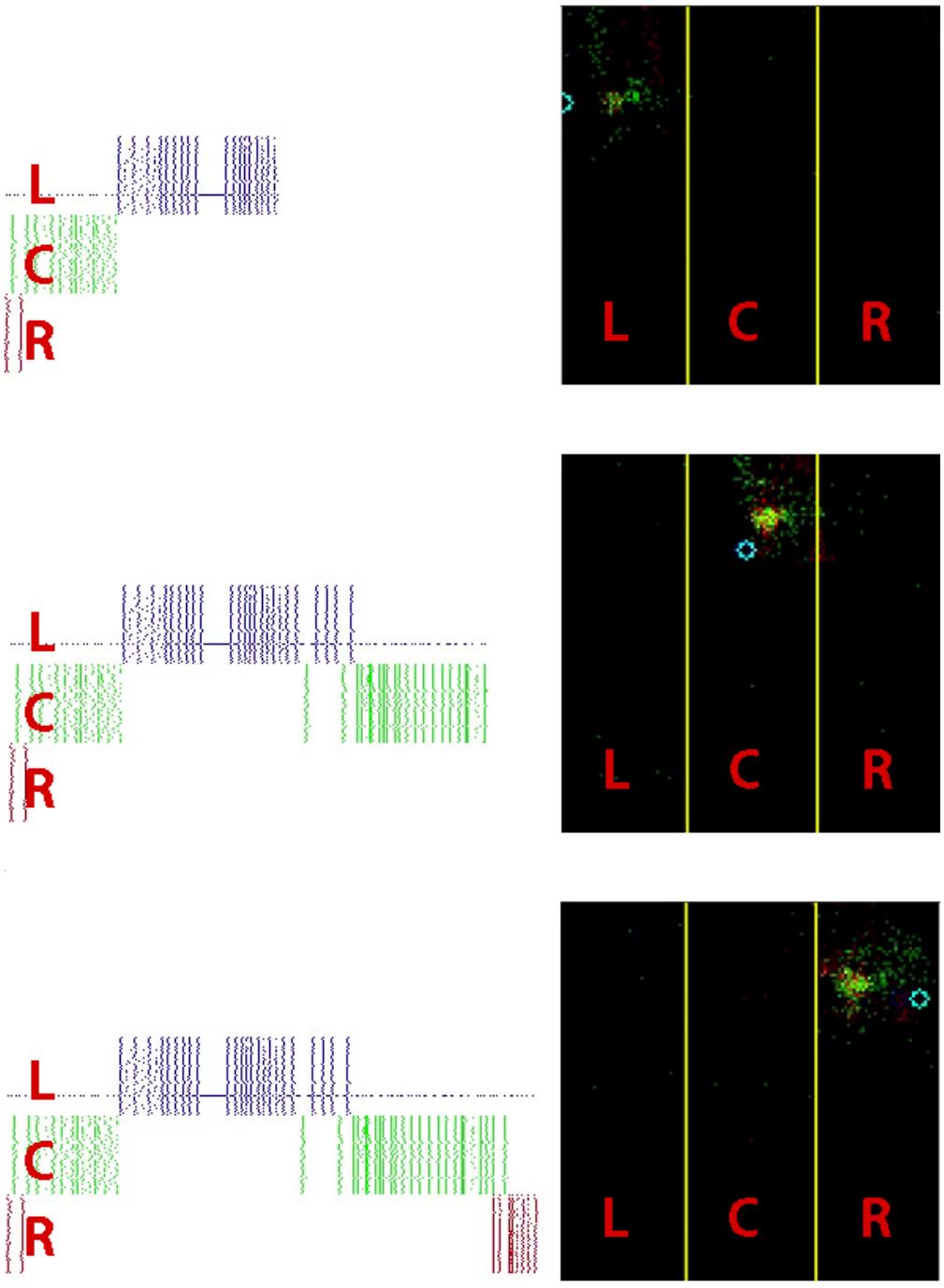

Figure 21. Screenshots of the DVS events and the spiking neurons in the webAEX interface. The black window on the right displays the events from the DVS after dropping $90\,\%$ of them. The red, blue and green stripes on the left represent the corresponding spiking neurons on the ROLLS. The cyan circle in DVS events window shows the position of the COM estimated by the spiking neurons. The images from top to down correspond to an flashing LED that is slowly but steadily moved in front of the DVS camera from right to left

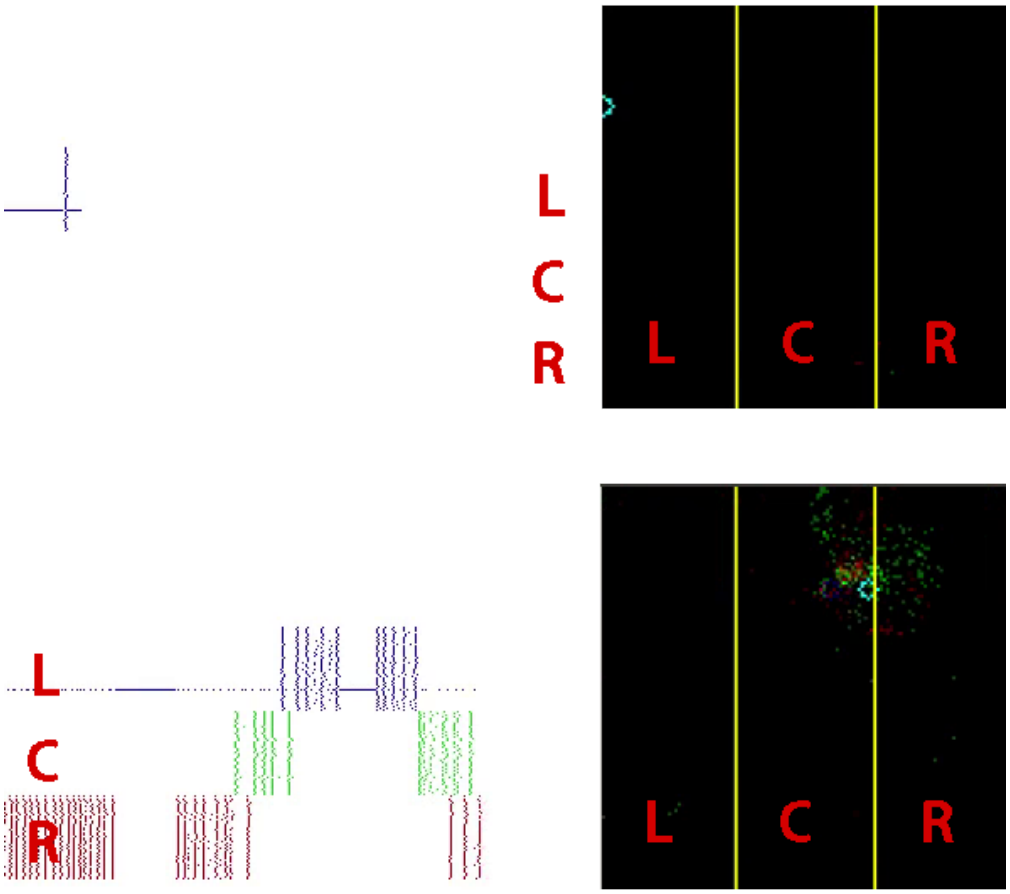

Figure 22. Screenshots of the DVS events and the spiking neurons in the webAEX interface. The black window on the right displays the events from the DVS after dropping $90\,\%$ of them. The red, blue and green stripes on the left represent the corresponding spiking neurons on the ROLLS. The cyan circle in DVS events window shows the position of the COM estimated by the spiking neurons. The top picture shows the ROLLS response when no input is present and o neurons are stimulated. There is one neuron firing continuously though. The bottom picture shows the ROLLS response when the LED is rapidly moved from left to right in front of the camera. In the window on the right with the DVS events a significant overlap between the center and right region is visible. The overlap of the spiking neuron populations is however quite small.

4 Conclusion and Outlook

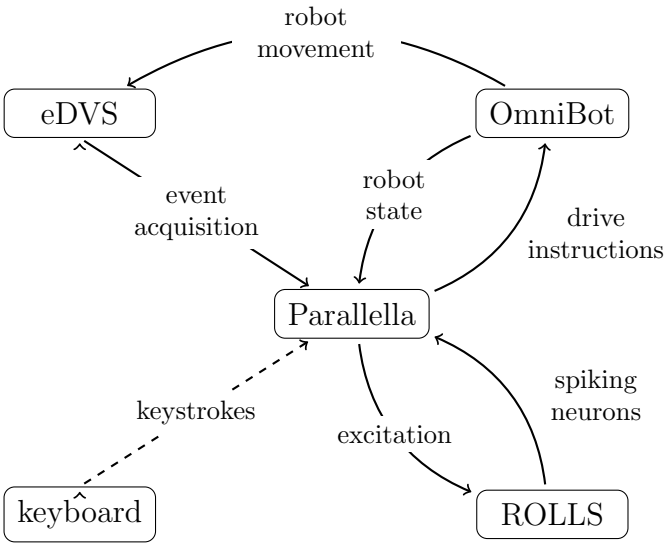

Figure 23. Schematic showing the different components that form the neuromophic computing platform explored in this project and the interactions among them.

In this project a DVS camera and the ROLLS neuromorphic chip were combined with a mobile robotic platform to demonstrate an embedded neuromorphic computing platform. This platform can be either controlled by keyboard input of an human operator or act autonomously by processing the output of spiking neurons from the ROLLS neuromorphic chip. Each of these components can be interfaced and controlled with a set of flexible interfaces developed in this project that capable of asynchronous event handling by relying in the the observer programming pattern. These interfaces enable the rich interactions depicted in figure 23.

A tool for ground truth acquisition was developed using these interfaces, providing real-world data serving as input for simulations or validation of simulations as well as training data for neural networks.

A second application making the robot follow a blinking LED, demonstrates that this mobile robotic platform can be effectively controlled by the spiking neuron output of the ROLLS chip and makes this platform a staging point for future experiments.

4.1 Future work

The interfaces to the different components of the setup is work in progress, especially for the robots. The OmniBot for example is developed by the NST group in Munich who continuously upgrade the firmware providing new features that can be included into the code. The are still some features not yet implemented in the ground truth acquisition tool, like the curved robot trajectories set by the key combinations defined in figure 9 on page 11 or gathering IMU data. Another interesting feature might be some filters for the incoming DVS event streams, currently simply 90 % of the incoming events are dropped, but there are certainly smarter filter around. Currently some parts of the code are Linux specific and one could spend some effort on making the code platform independent by using an event-based framework as Qt (see also section B on page 32 for some unfinished work in that direction).

There are also room for improvements on the LED tracker application. Currently the position of the LED with respect to the DVS camera is estimated on the Parallella, but one could use a neural network, such a Winner-Take-All network instead that can directly supply the necessary drive controls. Acquiring the position of the LED also needs to be made more robust or extended to other features. It was also planned to include obstacle avoidance by evaluating the lower part of the eDVS retinal space. Some code in that direction is available from the CapoCaccia workshop in the gitlab repository. An other issue is the horizontal resolution of the LED tracker, that

is reduced to 3 at the moment but can bi increased to take advantage of the full 128 pixel resolution of the camera. The controllable biases of the ROLLS neuromorphic chip can also be further tuned. In figure 21 on page 23 for example one neuron is always firing, even in absence of stimulation. This problem can be addressed by identifying the index of the neuron and changing which bias defines the time constant, usually by enabling `TAU2` for all the other neurons, and shutting off the troublesome neuron by enabling `TAU1` for it and setting the bias to a high value.

Acknowledgments

At this place I want to thank Prof. Indiveri for giving me the opportunity to do this project in his group. I also want to thank Yulia Sandamirskaya and Mortitz Milde for supervising this project and the many interesting discussions and the helpful advice. Also many thanks to Jonathan Binas for technical advice on the Parallellla and all the other people at INI for advice and help.

Parts of the code were developed during the CapoCaccia Cognitive Neuromorphic Engineering Workshop. This includes combining all the interfaces into the commandline tool for ground truth acquisition, the DVS synchronizer as well as an example for obstacle avoidance with spiking neuron output from the ROLLS chip that was adapted for the LED tracker.

References

[1] Bartolozzi C, Rea F, Clercq C, Fasnacht DB, Indiveri G, Hofstätter M, Metta G. Embedded neuromorphic vision for humanoid robots. In: *CVPR 2011 WORKSHOPS*. 2011; pp. 129–135.

[2] Lichtsteiner P, Posch C, Delbruck T. A 128×128 120 dB 15 *mus* Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE Journal of Solid-State Circuits*. 2008;43(2):566–576.

[3] Qiao N, Mostafa H, Corradi F, Osswald M, Stefanini F, Sumislawska D, Indiveri G. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. *Frontiers in Neuroscience*. 2015;9.

[4] The Parallella Board. `https://www.parallella.org/`. Accessed: 2016-05-30.

[5] User Guide - eDVS4337 (embedded Dynamic Vision Sensor) and PushBot. `http://inilabs.com/support/hardware/edvs/`. Accessed: 2016-05-30.

[6] Conradt J, Cook M, Berner R, Lichtsteiner P, Douglas R, Delbruck T. A pencil balancing robot using a pair of AER dynamic vision sensors. *2009 IEEE International Symposium on Circuits and Systems*. 2009;.

[7] Lichtsteiner P, Posch C, Delbruck T. A 128×128 120 dB 30 mW asynchronous vision sensor that responds to relative intensity change. 2006; pp. 508+491. Cited By 47.

[8] Accessing the device manually. `http://inilabs.com/support/hardware/edvs/#h.2v48aal08cim`. Accessed: 2016-05-30.

[9] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Abstraction and reuse of object-oriented design. In: *ECOOP '93*. Springer-Verlag. 1993; pp. 406–431.

[10] Observer pattern. `https://en.wikipedia.org/wiki/Observer_pattern`. Accessed: 2016-05-30.

[11] *Documentation* `omnibot-lib`. Https://code.ini.uzh.ch/mmilde/omnibot-lib.

[12] OmniRob: Mobile Robot Base with Robotic Arm for Manipulation Tasks. `http://www.nst.ei.tum.de/projekte/edvs/`. Accessed: 2016-05-30.

[13] Ribeiro AF, Moutinho I, Silva P, Fraga C, Pereira N. Three Omni-Directional Wheels Control On A Mobile Robot. IEEE. 2004; .

[14] Build an easy holonomic "Kiwi drive" robot platform that moves instantly in any direction. `http://makezine.com/projects/make-40/kiwi/`. Accessed: 2016-05-30.

[15] Bartolozzi C, Mitra S, Indiveri G. An ultra low power current-mode filter for neuromorphic systems and biomedical signal processing. In: *2006 IEEE Biomedical Circuits and Systems Conference*. 2006; pp. 130–133.

[16] *Manual MN256R1*. Internal communication.

[17] Qt documentation. `http://doc.qt.io/`. Accessed: 2016-06-06.

Table 4. Possible inputs for the argument <population> of aertest_modif and the corresponding stimulated neuron populations.

<population>	Neuron index range
1	0-50
2	51-100
3	101-150

A LED Tracker Tutorial

In this tutorial it is assumed that the serial-to-USB hub with the DVS camera and the robot are connected on the micro USB slot of the Parallella and the Parallella itself is connected with a Ethernet cable to the INI network and can be remotely accessed with SSH:

```
ssh -X root@<host>
```

Where host is the IP address or the name of the Parallella in the INI network. In our case the Parallella is registered as `parallelly`. The ROLLS chip is connected with the corresponding breakout board to the Parallella. After powering up the Parallella and logging, the first thing to do is starting the webserver with the bash script `server`:

```
server
```

This also initializes the ROLLS chip in a clean working state by configuring the initial biases. One can check if the server is running by typing in a web browser while connected to the INI network:

```
<host>:9000
```

If the system is running you should see a window similar to figure 20 on page 22. You can then proceed to load your experiment in the leftmost tab and load a set of saved biases (for the LED tracker `LED_tracker_27_05_b`, the relevant values from the IF and NPA tabs are also documented in table 5 and 6) or change them in the two other tabs. New experiments can be generated by adapting existing experiments from the directory

```
cd /var/www/aex/static/
```

Beware, if you try to load biases while stimulating the neurons, the system may crash. Changing biases in the first tab seems not to be compromised. You can observe the effect of different biases by using the `aertest_modif -s <population>` utility located in

```
cd /home/parallella/paex-dist-cc/aertest/
```

This utility can continuously stimulate neuron populations of 50 neurons in the index range 0 to 150. The possible values of the argument and the corresponding stimulated neuron populations can be found in table 4.

The LED tracker example is located in the folder:

```
cd /home/parallella/code/MadBot_Michel/USB_robot/test/build
```

And can be started by invoking

```
./robotloop
```

N.B. Sometimes the neuron populations seem indifferent to an external stimulus. In that case you may stimulate them from the outside with `aertest_modif` as a first diagnostic measure in conjunction with the webAEX tool.

A.1 Checklist for experiments with the ROLLS on the Parallella

- Check if all the devices are connected
  - DVS and Robot over micro USB to the Parallella.
  - Parallella over Ethernet or a WiFi dongle[1] to the INI network.

[1] If you use the WiFi dongle, make sure the necessary drivers are installed and working

- Remote access the Parallella
- Start the webserver
- Open the webserver in your browser
- Load your experiment
- Load your biases
- Start your experiment
- If your experiment does not work and it did in the past, check the neurons with `aertest_modif -s <population>`

Table 5. Biases as configured in webAEX tool. The values are in the same format as needed for the web tool and can be directly entered. The current is specified as maximum range (column Current) and the fraction of the current supplied (column Current Fraction). Maximum current is supplied when current fraction is set to 255. If we set `NPA_PWKL_P` to 6.5n and 45 means we are setting this bias to effectively $45/255 \cdot 6.5 = 1.15\,\mathrm{nA}$

Bias name	Current	Current Fraction
NPA_PWKL_P	6.5n	45
NPA_WEIGHT_STD_N	15p	0
NPA_WEIGHT_EXC_P	24u	160
NPA_WEIGHT_EXC0_P	24u	120
NPA_WEIGHT_EXC1_P	3.2u	150
NPDPIE_THR_P	820p	255
NPDPIE_TAU_P	6.5n	10
NPA_WEIGHT_INH_N	24u	100
NPA_WEIGHT_INH0_N	24u	255
NPA_WEIGHT_INH1_N	24u	255
NPAPDPII_TAU_P	24u	255
NPDPII_THR_P	15p	0

Table 6. Biases as configured in webAEX tool. The values are in the same format as needed for the web tool and can be directly entered. As in table 5 the current is specified as maximum range (column Current) and the fraction of the current supplied (column Current Fraction).

Bias name	Current	Current Fraction
IF_RST_N	15p	0
IF_BUF_P	15p	0
IF_ATHR_N	15p	0
IF_RFR1_N	15p	3
IF_RFR2_N	105p	60
IF_AHW_P	15p	0
IF_AHTAU_N	6.5n	193
IF_DC_P	15p	0
IF_TAU2_N	105p	105
IF_TAU1_N	105p	105
IF_NMDA_N	15p	0
IF_CASC_N	15p	0
IF_THR_N	15p	4

B Started work on an unfinished GUI using Qt

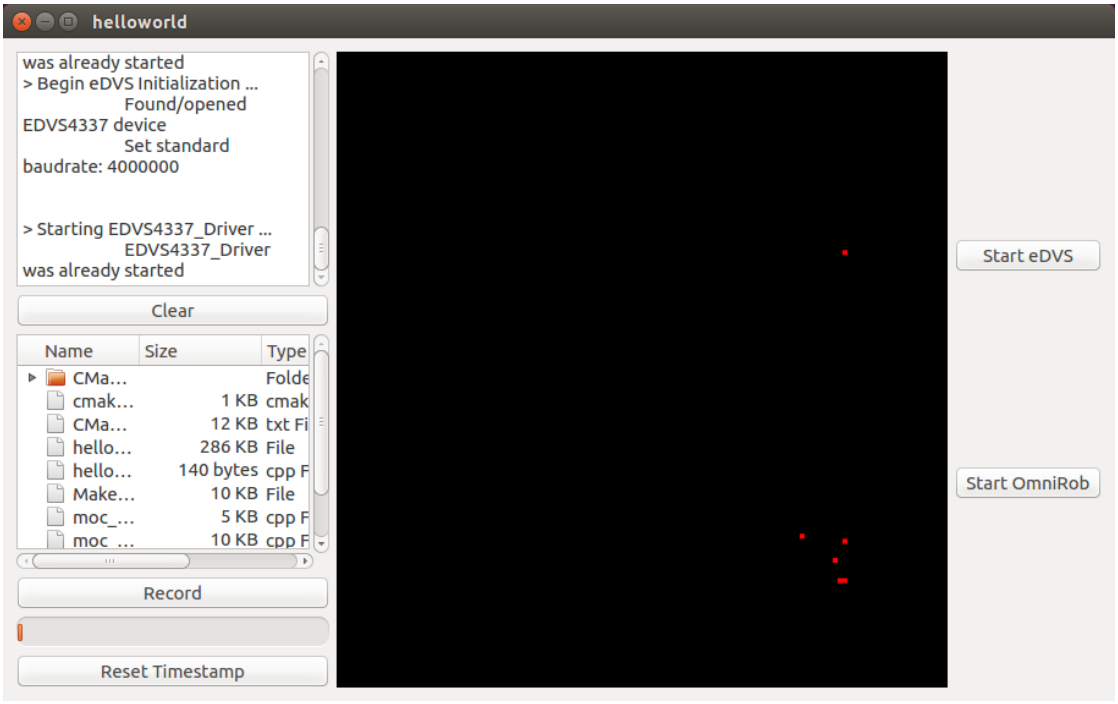

Figure 24. Screenshot of the main window of an unfinished tool for ground truth acquisition using Qt.

Qt is an mature, open-source, cross-platform, event-driven framework that can be deployed to various software and hardware systems, even on embedded systems[17]. Qt extends the C++ programming language by `signals` and `slots`, that implement the enhanced versions of the Observer pattern without any programming overhead. Considering requirements such as asynchronous data handling and a light-weight solution that can be deployed to embedded systems, Qt offers an attractive framework for this project. Figure 24 shows a screenshot of a simple and still unfinished alternative tool for ground truth acquisition with a GUI. The black window displays the events parsed from the DVS data stream. Different connected components can be started with the buttons on the right. On the left different widgets provide diagnostics about the system that are usually printed on the command line. The status bar below the record button shows the free capacity of the circular array buffering the data before they are written to the disk. The treeview above the record button displays the current log files. The development of this application was also aimed at addressing thread-safety issued that are not yet handled by the current command-line tool, such as concurrent access to the shared variables for logging and parsing from the streams. The code for this application can also be found in the gitlab repository (`https://code.ini.uzh.ch/mmilde/omnibot-lib`) in the folder `Qt_viewer`.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

An embedded neuromorphic computing platform for cognitive agents

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Frising	Michel

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 10.6.2016	Michel

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.