Faculty of Physics and Astronomy
University of Heidelberg

Diploma thesis

in Physics

submitted by

Eric Müller

born in Heidelberg

September 2008

Operation Of An Imperfect Neuromorphic Hardware Device

This diploma thesis has been carried out by Eric Müller at the
Kirchhoff Institute for Physics
University of Heidelberg
under the supervision of
Prof. Dr. Karlheinz Meier

Operation Of An Imperfect Neuromorphic Hardware Device

This thesis presents the work done towards the characterization and improvement of the operation of an imperfect neuromorphic hardware device. The hardware system and the utilized software framework are described. The developed methods for testing and specifying various chip features and imperfections are introduced, the results and a set of developed solutions are shown. These include an automated test suite for the digital part of the utilized chip, specification of its parameter stability, of noise types, of its long-term synaptic plasticity and of a specific problem with the readout of internal observables. Improvements of the software framework and of its development flow are presented, including a speedup of the build process, code profiling techniques and a graphical front-end for interactive operation of the hardware system. The presented work provides important insights into the functions and malfunctions of the utilized chip, which will eventually help improve its future revisions. The contributions to the software environment increase efficiency and transparency of the development flow and facilitate an experimental usage of the chip.

Betrieb eines unvollkommenen neuromorphen Hardwaresystems

Die vorliegende Arbeit befasst sich mit der Charakterisierung und Verbesserung eines noch unvollkommenen neuromorphen Hardwaresystems. Zunächst werden das Hardwaresystem und die verwendete Softwareumgebung beschrieben. Es werden Methoden vorgestellt, verschiedene Eigenschaften des Mikrochips zu testen und Fehlverhalten zu spezifizieren. Ergebnisse der hier entwickelten Methoden und Lösungsansätze werden präsentiert. Diese beinhalten automatisierte Testroutinen für den Digitalteil des verwendeten Chips, die Spezifikation der Parameterstabilität, des Rauschens, der synaptischen Langzeitplastizität und ein Problem mit der Auslese interner Größen. Desweiteren werden Verbesserungen zur Softwareumgebung und des Entwicklungsprozesses präsentiert. Dazu gehören eine Beschleunigung des Kompilierungsprozesses, eine Laufzeitanalyse und eine graphische Oberfläche für die interaktive Anwendung des Hardwaresystems. Die präsentierte Arbeit liefert Erkenntnisse über Wirkungsweise und Fehlverhalten des verwendeten Chips, die zu Verbesserungen in zukünftigen Revisionen beitragen können. Die Beiträge zur Verbesserung der Softwareumgebung erhöhen die Effektivität und Transparenz des Entwicklungsprozesses und erleichtern die Benutzbarkeit des Chips.

Contents

1 Introduction **1**
- 1.1 Neuroscience And Technology . . . 1
- 1.2 The FACETS Project . . . 3
- 1.3 Outline . . . 4

2 The FACETS Hardware **5**
- 2.1 Chip Overview . . . 6
- 2.2 Neuron Model . . . 7
- 2.3 Synaptic Plasticity . . . 7
  - 2.3.1 Short-Term Plasticity . . . 7
  - 2.3.2 Long-Term Plasticity . . . 8
- 2.4 Towards Stage 2 . . . 10

3 Software Framework **11**
- 3.1 The Meta Language PyNN . . . 11
- 3.2 The PyNN.hardware module . . . 13
  - 3.2.1 Python Hardware Abstraction Layer . . . 13
  - 3.2.2 Low-level Software . . . 15
  - 3.2.3 Graph Model . . . 16

4 Experimental Investigation Of The FACETS Stage 1 hardware **19**
- 4.1 Assertion of Digital Functionality . . . 19
  - 4.1.1 Link Test . . . 19
  - 4.1.2 Parameter RAM Test . . . 21
  - 4.1.3 Event Loopback Test . . . 21
  - 4.1.4 Conclusion . . . 21
- 4.2 Characterization Of The Chip . . . 22
  - 4.2.1 Noise . . . 22
  - 4.2.2 Neuron Readout . . . 22
  - 4.2.3 Parameter Stability . . . 24
  - 4.2.4 Spike Timing Dependent Plasticity . . . 25
- 4.3 Workarounds for Chip Imperfections . . . 28
  - 4.3.1 Summary Of Main Problems . . . 28
  - 4.3.2 Neuron Resets . . . 28
    - Every nth Input Spike . . . 28
    - Fixed Rate . . . 29
    - Poisson Distributed . . . 30
    - Conclusion . . . 30
  - 4.3.3 Parameter RAM Update . . . 32
  - 4.3.4 Parameter Voltage Generator Calibration . . . 35

5 Improved Software Flow and Tools **37**
5.1 Software Project Management 37
5.1.1 Build Process 37
5.1.2 Source Code Management 37
5.2 Korescope 38
5.3 Code Profiling 39
5.4 WinDriver 39

6 Discussion & Outlook **41**

A Appendix **43**
A.1 Source Code Locations 43
A.1.1 General 43
A.1.2 Further Contributions 43
A.1.3 Documentation 44
A.2 Example: KoreScope 45
A.3 SciPy.Weave Example 46

Bibliography **47**

1 Introduction

1.1 Neuroscience And Technology

To understand the most sophisticated information processing technology nature has come up with after billions of years of evolution – the human brain – is one of the fundamental questions of mankind. Scientists of nearly all disciplines have been involved in researching the human brain for centuries. Until today, many details have been investigated and are now well understood. However, many unanswered questions remain. Large efforts are made in all three major directions of neuroscientific research: microscopic studies to investigate the dynamics and mechanisms of single cells and connections, research at macroscopic scales to investigate the activity of whole brain regions and brain anatomy, and complexity research to investigate functionality, information processing and self-organization arising from connectivity, from complexity and from specific network architectures.

As in nearly all natural sciences, technological advances have always massively boosted neuroscientific research: For example, at the end of the 1970s the *patch clamp* technology provided revolutionary possibilities to measure single or multiple ion channels in neural cells. Since the 1980s neuroscientific modelers have been using computer simulations in order to gain new insights into both single cell dynamics and network behavior. Still today, as traditional computer systems are getting more and more powerful, new facets of research emerge, for example utilizing cluster technology that allows the simulation of increasingly larger networks. In the late 1980s, improvements in CMOS[1] VLSI[2] technology made it possible for a new field of neuroscientific modeling to emerge: neuromorphic hardware. This field, too, has received significant benefit from several improvements in the utilized technologies facilitated major improvements. Today, accelerated[3] neuromorphic hardware implements sophisticated cell models.

But it is not only neuroscience benefiting from technological improvements – the opposite direction is true as well. The ongoing research within this challenging field has always always pushed the employed technologies farther and developed new ones. Physiological access methods, pharmacological agents, simulation architectures and paradigms, micro-electronic engineering – studying the brain inspires and requires novel approaches that often find useful implications for other fields.

Of course – in addition to the pure academic interest – one major hope is that the gained knowledge about the brain will help understand mental disorders and develop effective therapies. Furthermore, from the technological point of view, new computational paradigms may emerge that could lay the foundation for completely new computer architectures. Brain-machine interfacing has evolved quite far already, paving the way for future neuro-prostheses and maybe even for various human enhancement technologies.

[1] Complementary Metal-Oxide-Semiconductor
[2] Very-Large-Scale Integration
[3] Speedup with respect to biological real-time of up to 10^5.

Many aspects of neuroscientific research are very technical – and so is the focus of this thesis. It is a work on the study and development of technologies for neuroscience. It describes the work done on enhancing the software controlling a neuromorphic hardware device, including the development of appropriate methods and tools, and on the specification of the hardware system's imperfections. Every newly developed technology exhibits imperfections along the way towards a satisfying state, and efficiently dealing with imperfections is an important scientific competence.

The following short survey shall roughly describe the status quo of the major fields in neuroscience and their most important techniques of knowledge acquisition.

The human brain comprises about 10^{11} *neurons*[4], each connected to thousands of its neighbors by dendrites and axons – via so-called *synapses*. *Neurons* transmit information in the form of stereotypical electric signals called *spikes*. A region in the brain called the *cerebral cortex* has been identified as playing a key role in our ability to think, remember, communicate, empathize, and plan for the future.

Biology Biological measuring methods can be divided into non-invasive and invasive, *in-vitro*[5] and *in-vivo*[6] methods. Examples for non-invasive methods for recording brain activity are *Electro-Encephalography* (EEG) and *functional Magnetic Resonance Imaging* (fMRI) [*Huettel et al.*, 2004]. The *Patch Clamp* technique [*Kandel et al.*, 2000], *Voltage Sensitive Dyes* (VSD) [*Kerr et al.*, 2005], *Micro-Electrode Array* (MEA) [*Frey et al.*, 2007], *Local Field Potential* (LFP) [*Mehring et al.*, 2003] are invasive measurement methods.

Modeling To classify neural network models according to their computational units, a classification scheme suggested by *Maass* [1997] can be applied. These so-called *first generation* models are based on *McCulloch-Pitts neurons* [*McCulloch and Pitts*, 1943]. *Perceptrons* [*Rosenblatt*, 1958] belong to the same model generation. A characteristic property of these models is the binary in- and output.

A biological interpretation of neural networks consisting of *second generation* neurons is added by interpreting their continuous output as *firing rates* of biological neurons.

Adding biologically realistic information coding – spatio-temporal patterns of action potentials, in the following also called *spikes* – yields *third generation* models employing *spiking neurons* as computational units.

A standard model focusing on high biological accuracy is the *Hodgkin-Huxley* model [*Hodgkin and Huxley*, 1952]. Modern versions of Hodgkin-Huxley-type models add additional ion channel populations based on experimental data, and complex geometries of dendrites and axons based on microscopy data. More phenomenologically oriented models reduce complexity to allow for efficient large-scale numerical simulation of groups of neurons [7] and to facilitate mathematical analysis of neurons as well as network dynamics.

A model often utilized for large-scale simulations is the so-called *Leaky Integrate-and-Fire* (LIF) model (e.g. the conductance-based IF models in *Destexhe* [1997]). Figure 1.1 shows an illustration of neuron behavior upon the arrival of excitatory post-synaptic potentials (EPSP). Multiple post-synaptic potentials are temporally integrated and, as the membrane

[4] A cell type receiving and transmitting signals within the nervous system is called *neuron*.

[5] inside or on living tissue of a living organism

[6] outside of an organism

[7] For example, the Blue Brain Project (`http://bluebrain.epfl.ch`) uses 8192 processors of a Blue Gene Supercomputer to simulate up to 10^8 *simple* neurons or 10^4 *complex* neurons in real-time.

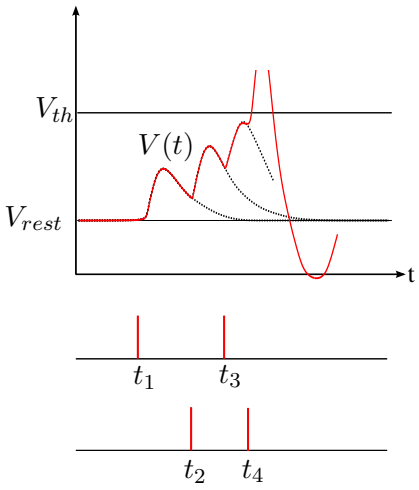


Figure 1.1: *Integrate-and-Fire* model: Membrane potential $V(t)$ depending on two pre-synaptic inputs. The first spike causes an conductance pulse to increase excitatory leakage. Thus, the membrane potential increases and an excitatory post-synaptic potential (EPSPs) can be seen on the voltage time course. After reaching a maximum value, the membrane potential declines due to a leak conductance. Integration of PSPs induced by incoming spikes can be observed upon arrival of the second and third spikes. A fourth spike lifts the membrane potential above threshold voltage V_{th}. An action potential (spike) is fired. Afterwards $V(t)$ undergoes a phase of hyper-polarization. Figure taken from [*Brüderle*, 2004] with friendly permission from Daniel Brüderle.

leaks towards resting potential, another spike triggers an EPSP lifting the membrane potential above the threshold voltage. A spike is generated, the neuron undergoes a phase of hyper-polarization and finally the membrane potential recovers towards resting potential. For some applications fixed threshold voltages are a oversimplification of the biological spiking behavior. Thus, various extensions allow for a more realistic spike initiation zone, subthreshold resonances or adaptation (e.g. *Adaptive Exponential Integrate-and-Fire* model in *Brette and Gerstner* [2005]).

Neuromorphic Hardware These models can also be implemented in a physical, typically silicon, form, mimicking the structure and emulating the function of biological neural networks. The first neuromorphic hardware models have been developed as far back as the 1980s [*Mead and Mahowald*, 1988; *Mead*, 1989]. Today, an active community develops analog or mixed-signal VLSI[8] models of neural systems [*Renaud et al.*, 2007; *Vogelstein et al.*, 2007; *Merolla and Boahen*, 2006; *Hafliger*, 2007; *Serrano-Gotarredona et al.*, 2006]. Physically emulating neural network models instead of numerically simulating them has two main features: the analog behavior and the inherent parallelism. Therefore, neuromorphic network models are typically exhibit a high scalability and can be operated in real-time or faster, independently of the underlying network size. Furthermore, since the analog circuits operate time-continuously, artifacts which may occur in discrete-time software simulators are avoided.

1.2 The FACETS Project

Since 2006, the author's group *Electronic Vision(s)* is member of the European joint research project *FACETS*[9]. The *FACETS* project consists of 15 partner groups, which employ scientists from various fields, including biology, mathematics, physics and engineering.

> *"The goals of the FACETS project is to create a theoretical and experimental foundation for the realization of novel computing paradigms which exploit the concepts experimentally observed in biological systems. The continuous interaction and scientific exchange between biological experiments, computer modeling and hardware*

[8] Very Large Scale Integration
[9] Fast Analog Computing with Emergent Transient States

> *emulations within the project provides a unique research infrastructure that will in turn provide an improved insight into the computing principles of the brain. This insight may potentially contribute to an improved understanding of mental disorders in the human brain and help to develop remedies."*[10]

Together with a team from the *Technische Universität Dresden*, the *Electronic Vision(s)* Group is responsible for designing and building a neuromorphic hardware system. This system implements a large number of emulated *spiking cortical neuron models*, i.e. the electrical behavior of these simplified neuron models is physically emulated by *CMOS*[11] circuits. Biologically realistic plasticity mechanisms are implemented in every single synapse. The systems can operate up to 10^5 times faster than their biological antetypes. Due to its inherent parallelism, this speedup is independent from the number of neurons operated. Thus, the system represents a potential platform for long-term or statistics-intensive experiments.

For the time being, the so-called *FACETS Stage 1* system is used as a prototype for the *Stage 2* system, which is under development. The *Stage 1* chip '*Spikey*' emulates a time-continuous leaky integrate & fire (LIF) neuron model (p. 7). 384 neurons and about 10^5 synapses (p. 7 et sqq.) are integrated into a single chip. In the *FACETS* project, the goal is a much higher integration density compared to the *Spikey* based system: the currently developed *Stage 2* system will comprise about $1.8 \cdot 10^5$ neurons and $4.2 \cdot 10^6$ synapses on a single wafer. Additionally, an improved neuron model including *spike-frequency adaption* and exponential voltage threshold [*Brette and Gerstner*, 2005] will be integrated.

An important aspect of the work done in the group is to make this hardware accessible to the interdisciplinary community of neuroscience. Therefore, the development of a project-wide unified meta-language *PyNN*, the integration of the hardware interface into PyNN (p. 11 et sqq.) and a detailed specification (p. 19 et sqq.) and calibration (e.g. p. 35 et sqq.) of the hardware are major focuses.

1.3 Outline

The present document is structured into two main parts:

- The description of the utilized experimental environment, i.e. the hardware system (chapter 2) and the software framework (chapter 3).
- The investigation of the utilized hardware system consisting of measurements of chip characteristics (sec. 4.2), workarounds for existing problems (sec. 4.3) and improvements made to software flow (chapter 5).

Contributions by the author to existing software, concepts and measurements are covered in chapter 4. Chapter 5 covers further technical contributions.

[10] `http://facets-project.org/`
[11] Complementary Metal-Oxide-Semiconductor

2 The FACETS Hardware

The *FACETS Stage 1* hardware system emulates a leaky integrate & fire, time-continuous neuron model [*Schemmel et al.*, 2004]. The conductance based synapses offer both a short-term plasticity and a long-term plasticity mechanism [*Schemmel et al.*, 2007, 2006]. Its design is based on existing phenomenological models [*Destexhe*, 1997]. Exploiting the possibility to implement very small capacitances and resistances in CMOS VLSI technology, high acceleration factors of up to 10^5 relative to biological real-time can be achieved. Almost $5 \cdot 10^4$ synapses and 384 neurons are integrated into a single chip. Figure 2.1 shows the experimental

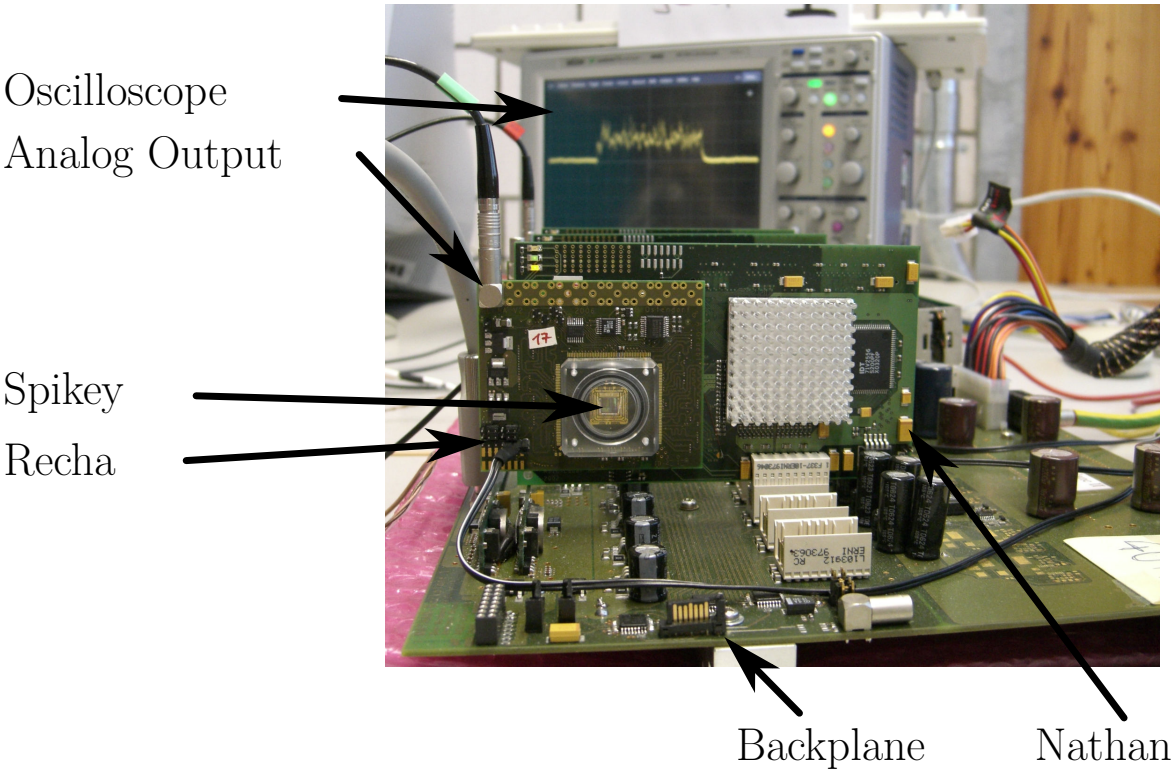


Figure 2.1: Photograph of the *FACETS Stage 1* hardware setup. *FACETS Stage 1* chip (called *Spikey*) on *Recha* support board and *Nathan* plug-in card; *Nathan* connected to a so-called *Backplane*; *Backplane* connected to host PC. In the background: Oscilloscope showing membrane trace.

setup: the *FACETS Stage 1* hardware chip (called *Spikey*) connected to a *Nathan* board carrying an FPGA[1] (below cooler) for experimental control. Up to 16 Nathans can be plugged into a *backplane*, which is connected to a host computer running *x86 GNU/Linux*.

[1] Field-programmable gate array

2.1 Chip Overview

The *FACETS Stage 1* hardware is produced using standard 180 nm technology with 1 poly-silicon and 6 metal layers and a $5 \times 5\,\mathrm{mm}^2$ die size. Two synapse arrays contain 49152 synapses (256×192) each and occupy most of the chip area. Located below, there are 192×2 neurons each containing a capacitance C_m that corresponds to the membrane capacitance. Three separate conductances model different ion channels (cf. figure 2.2 in the lower right corner):

- membrane leakage: g_{leak}, individually controllable for every neuron
- excitatory synapse conductance $g_x(t)$ and its reversal potential E_x
- inhibitory synapse conductance $g_i(t)$ and its reversal potential E_i

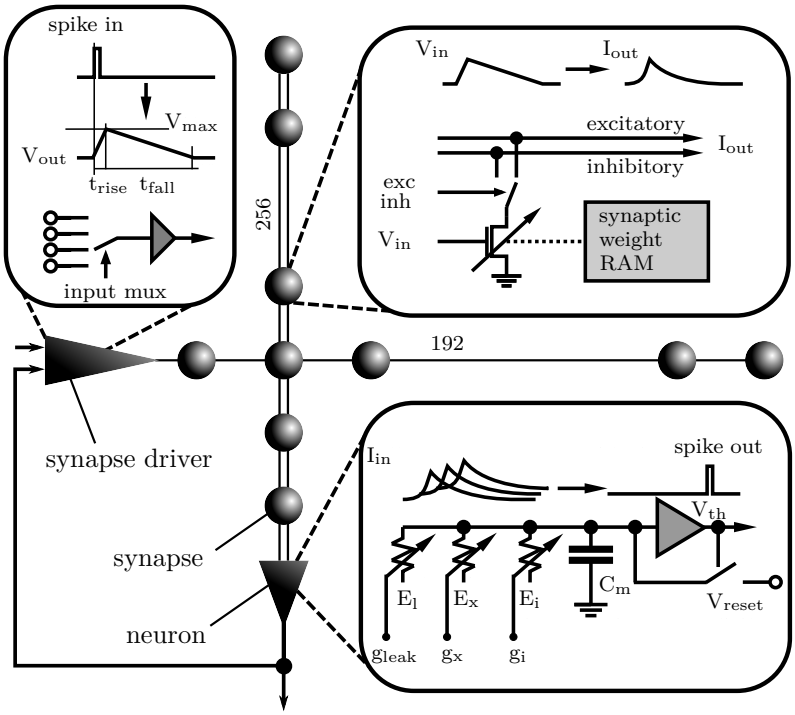


Figure 2.2: Operating principle of the *FACETS Stage 1* hardware device. Synapse drivers, synapses and neurons are marked by different boxes. A column corresponds to a neuron with up to 256 presynaptic inputs. Schematic based on *Schemmel et al.* [2004].

The leakage reversal potential E_l, the excitatory E_x and inhibitory E_i reversal potentials, as well as the threshold and reset voltages V_{th} and V_{reset} can be set for groups of 96 neurons. Synaptic weights are stored in a static RAM whose content is converted into a current by a 4-bit multiplying DAC in each synapse.

2.2 Neuron Model

The membrane potential V is determined by the following differential equation:

$$-C_m\frac{dV}{dt} = g_l(V-E_l)+\sum_k p_k(t)g_k(t)(V-E_x)+\sum_j p_j(t)g_j(t)(V-E_i) \quad (2.1)$$

$$g_{k,l}(t) = \omega_{k,l}(t)\cdot g_{k,l}^{max}(t) \quad (2.2)$$

The constant on the left side C_m defines the total membrane capacitance. The first summation term represents the contribution of the different ion channels that define the leak potential E_l. The reversal potentials for inhibitory and excitatory ion channels are determined by E_i and E_x. Excitatory synapses are covered by the second summation term, inhibitory synapses by the third summation term. The synaptic conductance $g_{k,j}$ is expressed as a product of a maximum conductance $g_{k,j}^{max}(t)$ and the synaptic weight $\omega_{k,j}(t)$ (cf. eq. 2.2). Short-term plasticity is modeled (cf. section 2.3.1) as individual synaptic open probabilities $p_{k,j}(t)$ that control the individual activations of the synapses [*Dayan and Abott*, 2001]. Long-term plasticity is incorporated into the model by assigning g_j and g_k weak time dependence $g_k = g_k(t)$ and $g_j = g_j(t)$ (cf. section 2.3.2).

2.3 Synaptic Plasticity

Synaptic Plasticity is the ability of a synapse to change its strength over time i.e. to change the weight of the connection between two neurons. The weight of a synapse describes the ability of the pre-synaptic neuron upon arrival of action potentials to influence the post-synaptic neuron (i.e. the effect on the post-synaptic potential).

2.3.1 Short-Term Plasticity

The *FACETS Stage 1* hardware implementation of *Short-Term Plasticity* [*Schemmel et al.*, 2007] is based on ideas developed in *Tsodyks and Markram* [1997] and *Markram et al.* [1998]: an absolute synaptic efficacy A_{SE} is introduced, distributed between a recovered (R) and an inactive (I) partition. Upon the arrival of an action potential (t_{AP}), a conductance pulse is generated, with a maximum g^{max} proportional to the fraction R of the recovered partition. After the transmission of the post-synaptic action potential, a fixed fraction U_{SE} of the recovered partition R is moved to the inactive partition I. A time-continuous reversal process reloads R, its time constant is t_{rec}. Figure 2.3 shows a NEST (cf. section 3.1) simulation of depressing and facilitating synapses. Incoming spikes are represented by bars. In the left panel, a depressing synapse is shown. With every incoming spike, its impact on the membrane is reduced. Depressing synapses obey:

$$\frac{dI}{dt} = -\frac{I}{t_{rec}} + U_{SE}\cdot R\cdot\delta(t-t_{AP}) \quad (2.3)$$

$$R = 1-I$$

$$g^{max} = A_{SE}\cdot R$$

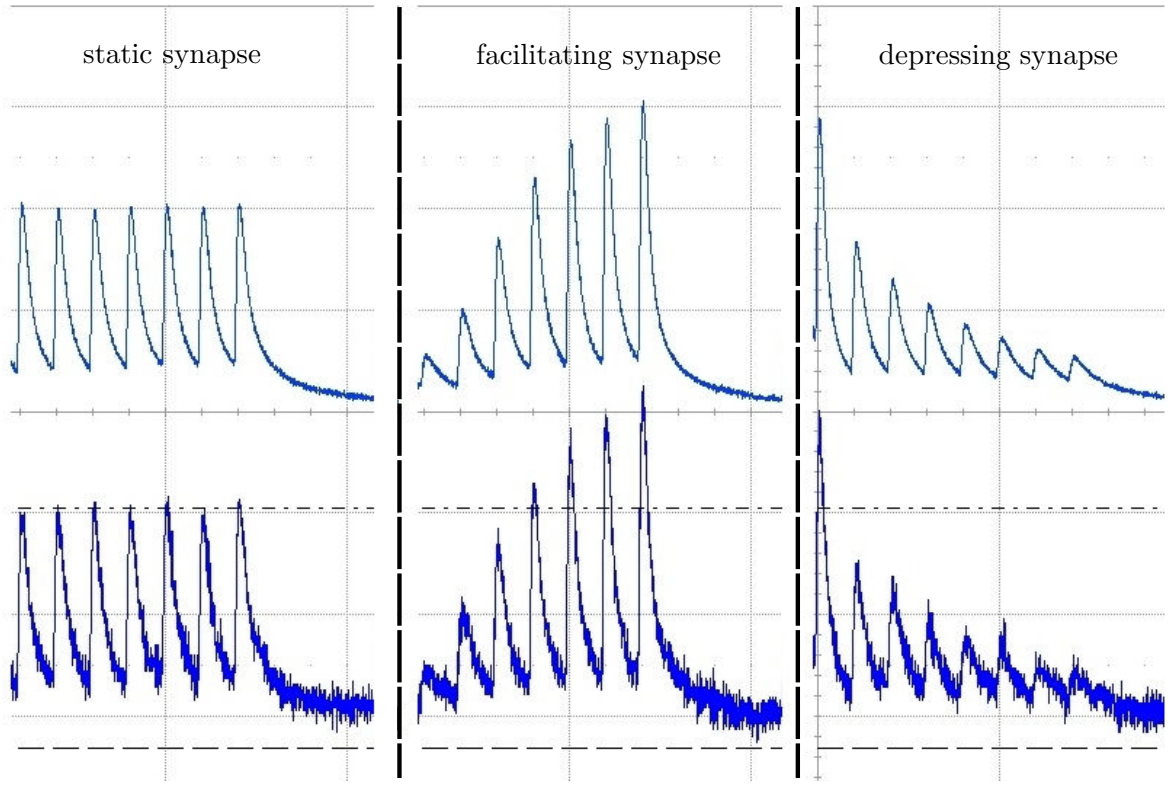


Figure 2.3: Equidistant spikes arriving at a hardware synapse cause a chain of post-synaptic potentials on the membrane. Demonstrated is the influence of a static (*left*), facilitating (*center*) and depressing (*right*) synapse configuration. In the upper half, the trace was averaged over 100 runs. With friendly permission of Johannes Bill [*Bill*, 2008].

To model facilitating synapses, R is replaced by I. Thus: $g^{max} = A_{SE} \cdot I$.

2.3.2 Long-Term Plasticity

The *FACETS Stage 1* hardware chip implements STDP[2] as a mechanism for long-term plasticity. It is based on physiological measurements presented in *Bi and Poo* [1997] and on corresponding models as for example described in *Song et al.* [2000]. For every occurrence of a pre- or post-synaptic action potential (with $\Delta t = t_{pre} - t_{post}$) the synapse changes the synaptic strength by a factor of $1 + F(\Delta t)$. F is called the STDP modification factor and is defined as follows:

$$F(\Delta t) = \begin{cases} A_+ \exp(\frac{\Delta t}{\tau_+}) & \text{if } \Delta t < 0 \text{ (causal)} \\ -A_- \exp(\frac{-\Delta t}{\tau_-}) & \text{if } \Delta t > 0 \text{ (acausal)} \end{cases} \tag{2.4}$$

Whereas A_+, A_-, τ_+ and τ_- are constant.

The *FACETS Stage 1* hardware chip implements STDP in every synapse [*Schemmel et al.*, 2006]. Since each hardware synapse contains its own STDP circuit, all correlation measurements are performed in parallel. Every synapse adds up these exponentially weighted

[2] Spike timing dependent plasticity

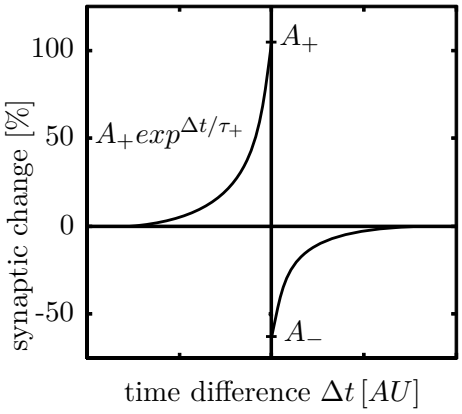


Figure 2.4: The STDP curve: pre-synaptic minus post-synaptic spike time yields time difference Δt (negative time difference implies causal, positive time difference implies acausal relationship between pre-synaptic neuron and post-synaptic neuron). Typical biological measurement routines use spike time differences of $0 \ldots 100\,ms$.

correlation measurements $F(\Delta t)$ individually for causal (pre-post) and acausal (post-pre) spike pairs. Thus eq. 2.4 transforms to:

$$\begin{array}{lll} F_c(\Delta t) = A_+ \exp(\frac{\Delta t}{\tau_+}) & F_a = 0 & \text{if } \Delta t < 0 \text{ (causal)} \\ F_a(\Delta t) = A_- \exp(\frac{-\Delta t}{\tau_-}) & F_c = 0 & \text{if } \Delta t > 0 \text{ (acausal)} \end{array} \tag{2.5}$$

F_c and F_a are added to $\sum F_c$ and $\sum F_a$ respectively. If the absolute difference between these sums exceeds a threshold $V_{threshold}$, a significant correlation is flagged. To mark a causal correlation $\sum F_c$ has to be larger than $\sum F_a$ and vice versa.

$$|\sum F_c - \sum F_a| > V_{threshold} \tag{2.6}$$

$$\sum F_c - \sum F_a > 0 \tag{2.7}$$

If eq. 2.6 evaluates to *true* and automatic updating is enabled, the weight is updated. This is done by replacing the current weight (4 bit resolution) ω of the synapse with the corresponding value obtained from either the causal or the acausal programmable look-up table. Which table gets used is determined by the result of eq. 2.7. That is, if $\sum F_c > \sum F_a$ is true, the causal look-up table is used and vice versa. Subsequently the accumulating values $F_{a,c}$ are reset. As long as eq. 2.6 evaluates to *false*, the summations continue.

However, the STDP controller works sequentially. Therefore, the weight update is delayed until the STDP controller processes the particular synapse. The following formula can be used to calculate the worst case[3] delay for updating the synaptic weights:

$$t_{update} = (2N_{\text{updates}/(6\times\text{row})} + t_{\text{row delay}}) \cdot 2t_{clk} \cdot N_{\text{rows}} \tag{2.8}$$

Every two clock cycles up to six weights can be updates. Each row access must be done twice, for causal and acausal measurements. Using the default operating parameters (at the time of writing) row access $t_{\text{row delay}}$ requires typically 50 cycles; t_{clk} is set to 200 MHz; speedup relative to biological real-time is 10^5. Thus $(2 \cdot 64 + 50) \cdot 2 \cdot 5\ \text{ns} \cdot 256 \approx 456\,\mu\text{s}$ are needed for updating all $192 \cdot 256$ synapses within a block. Considering a speedup of 10^5, this represents 45.6 s in biological time. This clearly limits the speed of synaptic weight adaptation via STDP in the hardware system. Therefore, depending on the experimental setup, biologically realistic learning behavior involving STDP cannot be guaranteed [*Morrison et al.*, 2008].

[3] Assumptions: all synapses are plastic and all synapse rows used (cf. figure 2.2)

2.4 Towards Stage 2

The next step towards the emulation of large neuronal networks is the FACETS Stage 2 hardware. Comparable to the *Stage 1* system, the hardware model consists of circuits implementing an *Integrate-and-Fire* neuron model and synapses providing short-term and long-term plasticity.

In contrast to the *FACETS Stage 1* hardware device, the hardware neuron models spike-frequency adaptation and an exponential voltage threshold. The new design is based on *Brette and Gerstner* [2005], a so-called *adaptive exponential integrate-and-fire* (aEIF) model. Furthermore, the *FACETS Stage 2* hardware system aims at a much higher integration density in comparison to *Stage 1* hardware: the challenging idea of scaling up the system to a whole silicon wafer.

A building block called *HICANN*[4] is being developed: it consists of the *ANNCORE*[5] which contains $1.2 \cdot 10^5$ synapses and up to 512 neurons and other support circuits. 64 of these neurons can be combined to increase the input count to $1.5 \cdot 10^4$.

Eight HICANN dies are combined to form an individual reticle on the wafer-scale system [*Schemmel et al.*, 2008]. For the time being, there will be enough space for ca. 44 reticles per wafer. Thus, $1.8 \cdot 10^5$ neurons and $4.2 \cdot 10^6$ synapses will fit onto a single wafer. Due to the high acceleration factor of about 10^4 compared to the biological real time, the necessary communication bandwidth in-between these *Analog Neural Network* blocks can exceed 10^{11} neural events per second, each encoding the transmission of an action potential from one source neuron to a set of target neurons [*Fieres et al.*, 2008].

To achieve this communication bandwidth we use wafer-scale integration. In this technology, the silicon wafers containing the individual chips are not cut into dies. Instead, the wafer will be post-processed by depositing and structuring an additional metal layer on top of the whole wafer interconnecting individual reticles directly on the wafer. This method provides the necessary connection density. Furthermore, support for multi-wafer interconnections is planned.

[4] High Input Count Analog Neural Network
[5] Analog Neural Network Core

3 Software Framework

The control of the *FACETS Stage 1* hardware, the so-called *Spikey* chip, is based on a multi-layer software framework. In the following sections the software layers are explained from top to bottom (from the experimenter's point of view):

PyNN is the Python package for simulator-independent specification of Neuronal Network models. It is an universal interpreter-based API[1] for simulator-independent specification of neural network models developed within the FACETS project as an unified simulation interface. [*Ensemble*, 2008]

PyNN.hardware The PyNN module for the *FACETS Stage 1* hardware system, which includes the hardware abstraction layer (PyHAL [*Brüderle et al.*, 2007]) and the so-called PyScope. The latter provides access to digitizing oscilloscopes with a TCP/IP network connection.

Low-level C++ The low-level code provides abstract C++[*Stroustrup*, 1997] interfaces for the existing hardware modules, thus providing an object-oriented interface to the *FACETS Stage 1* hardware device. Another module written in C++ is the core of *PyScope* which may be used to record analog data from an oscilloscope.

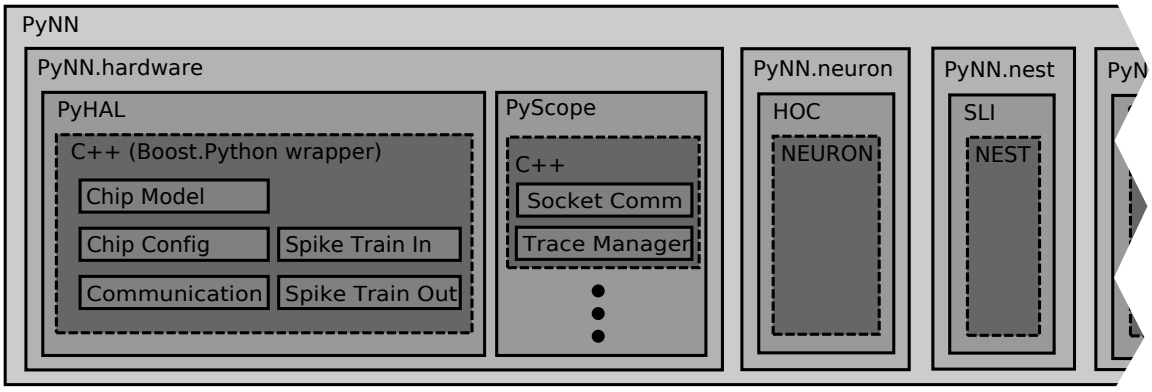


Figure 3.1: Software overview showing the software structure of the FACETS Stage 1 hardware system.

3.1 The Meta Language PyNN

The PyNN project [*Ensemble*, 2008] started in 2007 with the aim to develop both a generic API and bindings to different simulation environments using the Python programming language. For small networks, the low-level, procedural API provides functions to create (PyNN

[1] Application Programming Interface

function `create()`) neurons or spike sources, connect (PyNN function `connect()`) neurons to neurons or spike sources to neurons, record (PyNN function `record()`) spikes from neurons, record neuron membrane traces (PyNN function `record_v()`) and set (PyNN function `set()`) neuron, input or synapse parameters. For ease of use, a high-level API exists, that provides an object-oriented view of populations of neurons (PyNN class `Population`), sets of connections between populations called projections (PyNN class `Projection`). This allows for a simplified experimental setup, hiding the details and bookkeeping. Moreover, the *FACETS Stage 2* hardware system will require a thin-layered and parallelized interface to take advantage of the highly accelerated emulation of networks in the order of 10^6 neurons. Thus an object oriented approach using populations and projections to encapsulate experimental details is favorable.

Figure 3.2 shows the unifying functionality of PyNN. Thus, PyNN is a common language for setting up neuronal experiments on various back-end simulators. The simulator is specified once at the beginning of the script and can be replaced by any of the supported simulation back-ends at any time later. It is for this reason, that exchanging the simulation back-end requires just a change in a single line of source code (`import pyNN.SIMULATOR as pynn` is replaced by `import pyNN.ANOTHERSIMULATOR as pynn`). This allows for portable experiment specification, easy simulator comparison and data verification.

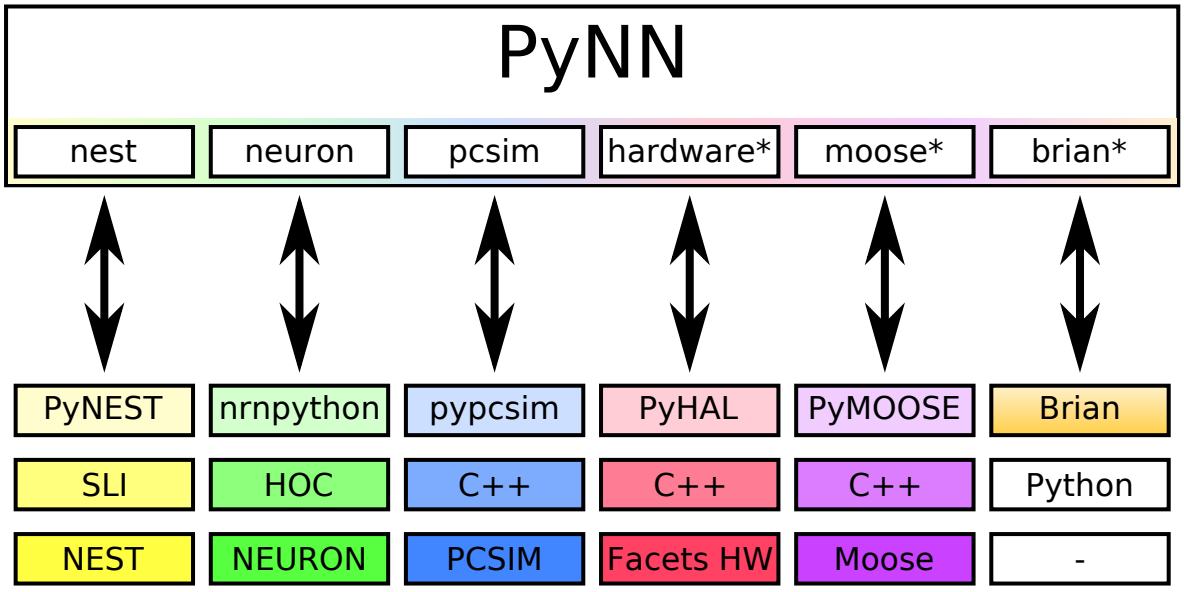


Figure 3.2: Schematic of the structure of PyNN. From top to bottom: PyNN modules, Python based interface, native interpreters and simulation kernels. Starred (*) PyNN modules are under development and presently not part of the distributed PyNN releases.

The following simulation back-ends are supported by PyNN:

nest The (*NEural Simulation Tool*) Initiative [*The Neural Simulation Technology (NEST) Initiative*, 2007] develops the NEST Simulator. PyNN scripts are translated via PyNEST, a Python based interface language, to the native SLI interpreter language. The NEST simulation kernel is written in C++.

neuron NEURON [*Hines et al.*, 2008] is a simulation environment for modeling individual

neurons and networks of neurons. PyNN support is based on a native Python interface to NEURON called *nrnpython*. Statements in HOC (native interpreter language of NEURON) are still supported from within nrnpython.

pcsim PCSIM is the successor of CSIM, the *Parallel neural Circuit SIMulator*. It is written in C++ with a primary interface to the programming language Python called *pypcsim*.

moose MOOSE is a simulation environment compatible with GENESIS. The native Python based interface to MOOSE is called PyMOOSE and a first step to interface it from PyNN was taken at FACETS CodeJAM Workshop #2[2] in May 2008.

brian Brian [*Brette and Goodman*, 2008] is a simulator for spiking neuronal networks written in Python. It aims to be easy to learn and use, while at the same time being highly flexible and easily extensible.

hardware The *FACETS Stage 1* hardware interface is based on *PyHAL*, a hardware abstraction layer written in the Python programming language. Low level code is written in C++. See section 3.2.

A short outline of NEST, NEURON, CSIM (predecessor of PCSIM) and GENESIS can be found in *Brette et al.* [2006].

3.2 The PyNN.hardware module

In order to make the *FACETS Stage 1* hardware system accessible by *PyNN*, a multi-layered approach was developed. On the top are the high-level PyNN modules (e.g. `Populations` or `Projections`), followed by the procedural PyNN functions (cf. section 3.1), descending through *PyNN.hardware* and its main component *PyHAL* to the low-level C++ code. PyNN.hardware itself is a very thin translation layer to map the PyNN API to the hardware abstracting PyHAL layer. In the following sections its primary component PyHAL and the low-level C++ code are introduced. A detailed schematic is shown in figure 3.3. The currently developed software framework shown in figure 3.5 is being prepared for the future *FACETS Stage 2* system.

PyNN is primarily[3] an API specification to enable and encourage researchers to write simulator-independent code. The creation of a new PyNN wrapper supporting a new simulator back-end requires creating functions and classes calling underlying existing code. In this case, the PyNN.hardware module uses the PyHAL infrastructure. Another part is the *PyScope* module, which supports remote control and acquisition of analog output from the chip using a specific oscilloscope type.

3.2.1 Python Hardware Abstraction Layer

PyHAL is the Python Hardware Abstraction Layer supporting the *FACETS Stage 1* hardware[4]. It provides access to the *FACETS Stage 1* hardware via Python.

[2] `http://www.neuralensemble.org/meetings/CodeJam2.html`

[3] By now, there are ongoing efforts to create generic high-level class implementations. Thus, implementing a new PyNN wrapper is easier, as only the simple procedural PyNN interfaces must be translated. Most high-level classes can fall back on the generic implementation.

[4] To simplify the transition to the *FACETS Stage 2* hardware, Stage 2-specific software requirements such as the *Graph Model* (cf. section 3.2.3) are being integrated.

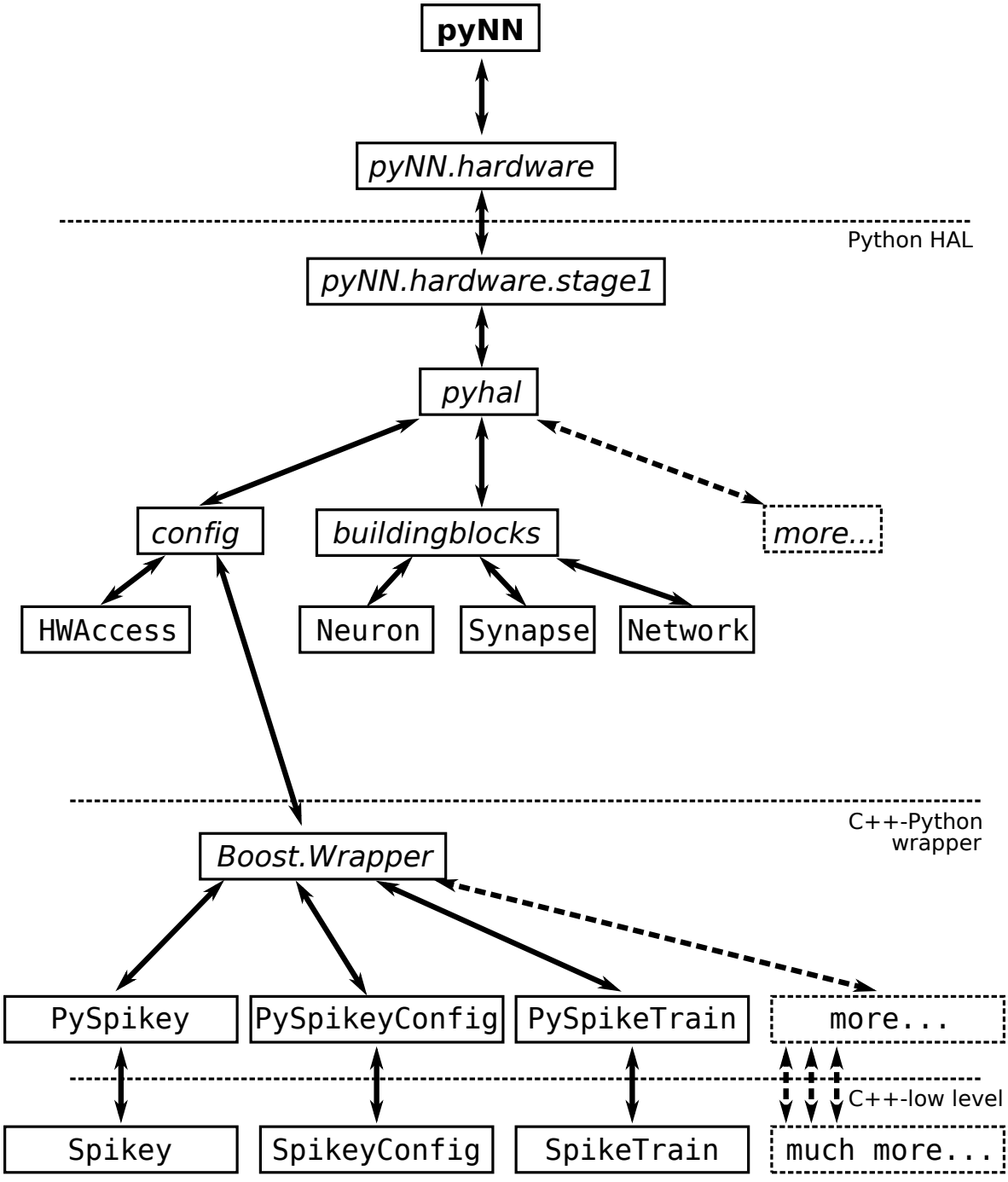


Figure 3.3: Schematic of PyNN.hardware (Stage 1). Modules (i.e. functions and classes grouped by namespace) are written in italics, classes in monospaced font.

PyHAL consists of two submodules called *buildingblocks* and *config*.

The former groups logical units into Python classes like `Neuron`, `Synapse` and `Network`. The `Neuron` class provides a modularized view of hardware neurons, holding essential hardware parameters like threshold voltages and a connection array. The `Synapse` class is designed for combining properties of the hardware implementation of synaptic plasticity(cf. section 2.3) mechanisms. The `Network` class is the top-level class, creating `Neuron`s, connecting `Neuron`s to external spike sources or each other. Connections between `Neuron`s are of type `Synapse` which is a class derived from `float` (the weight).

PyHAL's second submodule *config* provides the class `HWAccess`. The purpose of this class is to translate all model parameters (e.g. weights or threshold voltages) from the biological terminology and the mostly continuous and in principle unlimited values defined within the PyNN scope down to the discrete and device specific domain of hardware configuration parameters. For example, `discretizeWeight` converts the continuous model parameter *synapse weight* into a discretized 4-bit hardware weight. In order to keep the setup as close as possible to biological reality, weights differing from legal hardware values are evenly distributed between the two[5] discrete hardware values close to the original value. Another feature is the acquisition of analog membrane potentials using an external module called *PyScope*. However, the main functionality is to offer a mapping from the high-level PyNN network specification to the low-level hardware device in terms of connectivity and grouping of neurons/synapses sharing the same parameter sets. A good mapping algorithm reduces the need for manually tuning the PyNN script to support a larger network with higher connectivity (see section 3.2.3).

The PyHAL top-level joins these submodules into a conglomerate of intermediate interface functions for wrapping hardware access with upper-level PyNN. Therefore it mostly passes calls to the submodule *config* while administrative work is done using classes of the submodule *buildingblocks*.

As the *FACETS Stage 1* hardware system does not support inter-chip networking yet, the largest mappable network consists of 384 neurons. Network support will raise the need for sophisticated mapping algorithms. Furthermore, the upcoming *FACETS Stage 2* hardware[6] will require a fast and mapping-efficient[7] translation of a given PyNN network specification to the hardware system. Thus, an optimized software structure called *Graph Model* is under development. As figure 3.5 shows, almost all parts of the Python Hardware Abstraction Layer will be replaced by the so-called *Graph Model*. A basic introduction to the Graph Model is given in section 3.2.3.

For documentation related to this module see A.1.3 (p. 44).

3.2.2 Low-level Software

To call C++ from Python, multiple approaches exist: the maximum flexibility can be achieved with the *C API*[8], but this is rather complex and error-prone, as most work has to be done manually. A second method is *SWIG*[9] which is a tool to create wrapper code

[5] If the initial value lies outside the representable range of values, it is replaced by the largest (or smallest) allowed hardware value and a warning is issued.

[6] Up to $1.8 \cdot 10^5$ neurons and $4.3 \cdot 10^7$ synapses will be supported.

[7] To maximize network size and minimize synapses dropped.

[8] `http://docs.python.org/api/api.html`

[9] `http://www.swig.org/`

automatically for a number of different interpreted and compiled programming languages. The main disadvantage of SWIG is that as soon as complex[10] data structures are transferred between Python and C++, special interface code has to be written. A third wrapper interface is *Boost.Python*[11]. It is designed to wrap C++ interfaces as uninvasively as possible[12]. In most cases the underlying C++ code does not need extra editing in order to be wrapped by Boost.Python. Its support extends from references and pointers[13], function overloading, exception translation[14] to iterator translation. As Boost.Python satisfies all performance and memory usage requirements plus provides a convenient interface, it has been chosen as the wrapper interface for the *FACETS Stage 1* and *2* hardware system.

The wrapped low-level C++ layer[15] already provides a somewhat abstracted view of the hardware system. Hardware modules are modularized in an object-oriented manner but do not offer network-level abstraction.

The most important functions accessed by PyHAL are `config()` to configure the system, `sendSpikeTrain()` to send a spike train to the configured system and `recSpikeTrain()` to read a spike train from the chip. These functions are encapsulated into a single class called `Spikey`. This is the top-level class utilizing further support classes for communication with the chip (e.g. `SpikenetComm` and its descendant classes respectively) and configuration of different modules (e.g. `PramControl` for the parameter RAM setup).

For documentation of the low-level software see A.1.3 (p. 44).

3.2.3 Graph Model

In this context, the process of translating biological networks models as defined by PyNN into neuromorphic hardware configurations is from now on simply referred to as *mapping*. Connections in small networks can be represented as a *weight matrix*. Non-zero entries represent synapses, the column index corresponds to pre-synaptic neurons and the row index corresponds to post-synaptic neurons. If larger networks are involved and connectivity is sparse, arrays are needlessly memory consuming as their size grows quadratically with respect to number of neurons. Instead of that, sparse matrix techniques [*Pissanetzky*, 1984] may be used. But neuromorphic hardware systems add additional constraints such as limited connectivity, grouped units sharing a single parameter, differing latencies for off-chip connections and limited bandwidth. The time required to create a mapping between biological network specification and hardware configuration is another important point. A slow mapping process reduces the overall speedup the system can achieve.

In the face of the presently developed *FACETS Stage 2* hardware system (cf. section 2.4), the FACETS project partner group of the *Technische Universität Dresden* developed a *Graph Model* to solve these tasks. A graph comprises a set of vertices (or *nodes*) $V \neq \emptyset$ and a set of edges E that connect pairs of nodes. A hypergraph may also contain edges which connect more than two nodes ($E \subset V^N, N \leq \dim V$). A biological network consists of neurons and synapses, each of them characterized by a set of parameters. These types of units (e.g. neurons, synapses, parameters) are represented as nodes of the *BioModel*. Parameter assignments to

[10] non built-in data types

[11] `http://www.boost.org/doc/libs/1_34_0/libs/python/doc/index.html`

[12] Typically Boost.Python wrapper interface code requires a single line of source code to wrap a C++ function.

[13] Therefore most data structures can be transferred without copying data.

[14] C++ exceptions are translated to Python exceptions, thus allowing for more safe programming. Error conditions can be handled in a cleaner way.

[15] A more detailed overview can be found in [*Grübl*, 2007]

neurons are represented by directed edges, connections between neurons are represented by directed multi-edges (triples consisting of two neurons and a synapse). These two types of directed edges are called *named edges*. To indicate hierarchical dependencies undirected edges called *hierarchical edges* are added (e.g. connecting individual (*instantiated*) neurons to its base node). Besides these two edge types, a third one, called *hyper edge*, is used. These edges are created by a *mapping algorithm* and map biological units to hardware units. The *Hardware Graph* is constructed in a similar fashion: a top-level node points to *Wafer* nodes pointing to functional blocks et cetera. Figure 3.4 shows a simplified overview of the Graph Model after mapping a given biological network successfully.

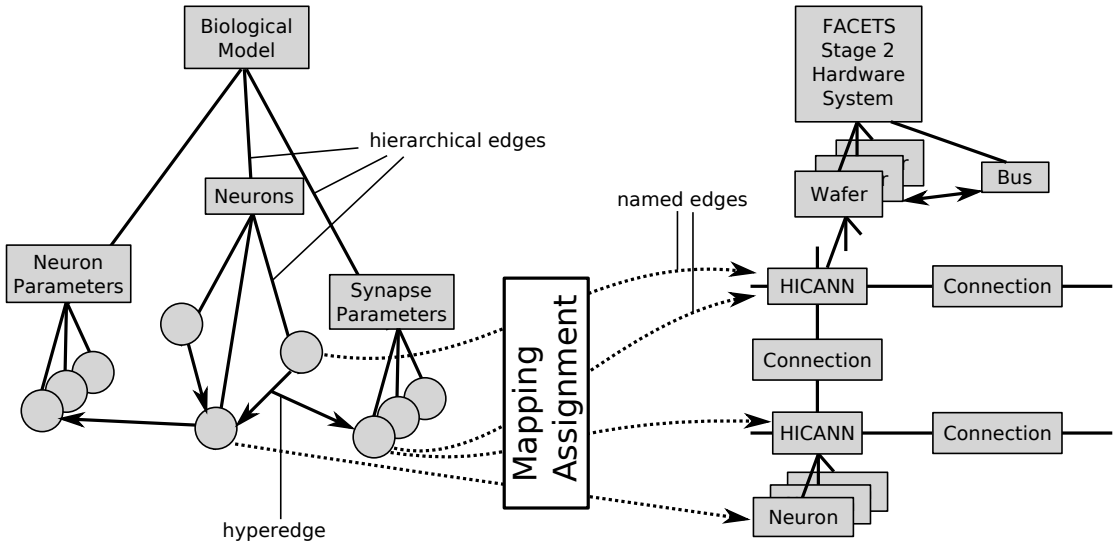


Figure 3.4: Graph Model: representations of the biological network (*left*) model and the hardware system (*right*). The mapping between these graphs is represented as dashed edges.

The primary task is to create a *good* mapping. *Good* may be characterized as: a) minimize neurons and synapses dropped (i.e. achieve a high utilization); b) minimize violations of parameters; c) minimize timing violations; d) realize low mapping time. These partly contradictory requirements form a multi-objective optimization problem that is hard to solve. An iterative mapping algorithm creates a mapping, calculates a score with the help of a *cost function* and tries to maximize the score. The details of the used mapping algorithms and cost functions are not described here in detail. A more detailed introduction to the Graph Model is given in *Wendt et al.* [2008], whereas parallelization efforts are discussed in *Ehrlich et al.* [2008].

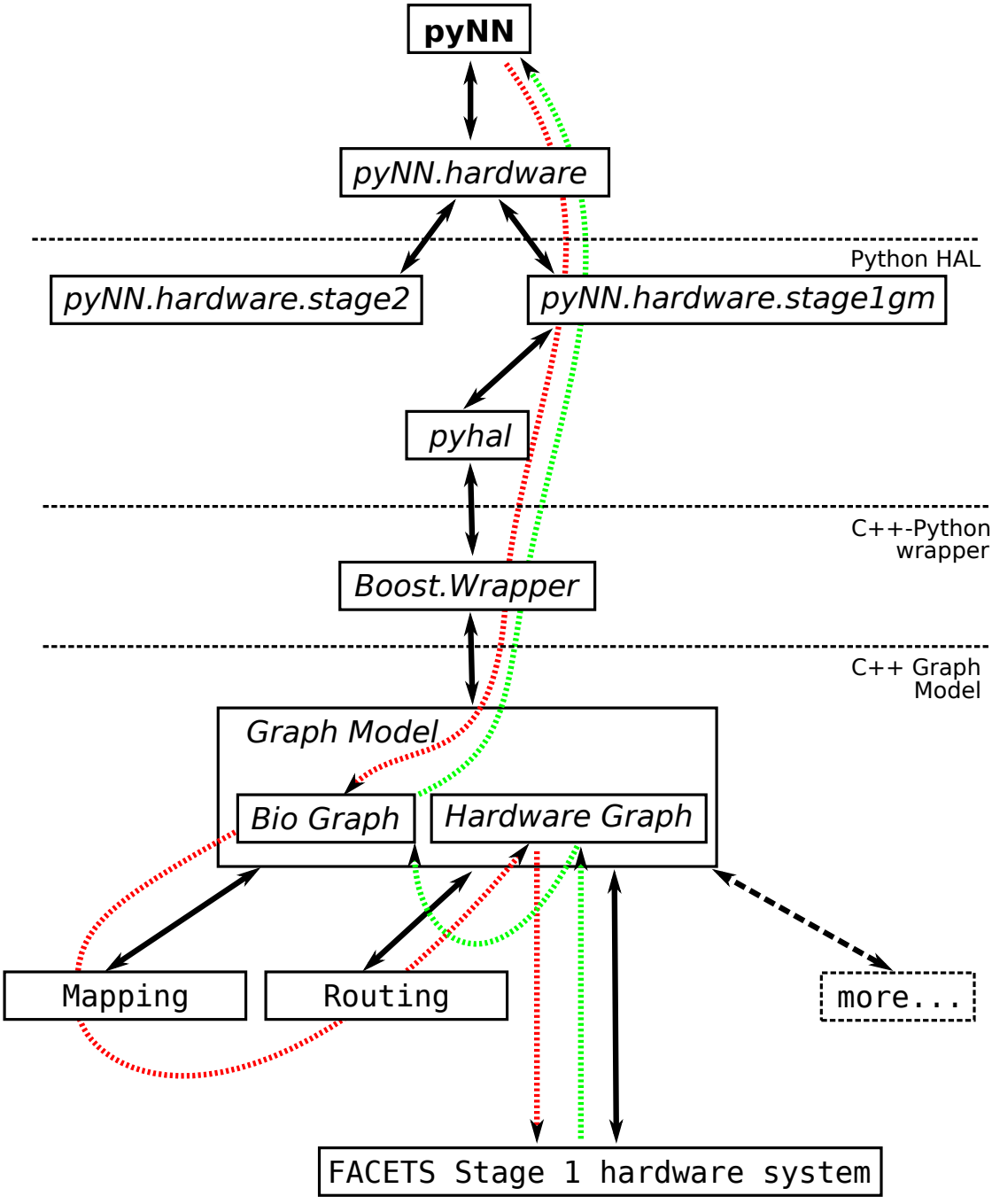


Figure 3.5: Currently developed schematic of PyNN.hardware (Stage 1) with *Graph Model* and future Stage 2 schematic. An exemplary call to `pynn.run()` is shown (dashed line). The configuration process is marked in red, returning data in green.

4 Experimental Investigation Of The FACETS Stage 1 hardware

4.1 Assertion of Digital Functionality

As neuromorphic hardware systems are getting increasingly complex, a multitude of potential sources of error arise inevitably. While most errors disrupt higher-level functionality and can thus be detected quite easily, some errors may stay unnoticed for a long time. Therefore, test charts are introduced to avoid hidden errors. As manual testing is cumbersome, automatic test procedures are essential. In the following, three tests of the PyHAL *hardware test* module which has been implemented by the author are described.

Low-level Tests Both the communication with the chip and its basic functionality can be tested by so-called *test modes*. These tests were developed originally for command line access by *Grübl* [2007, p. 133]. On top of this, the author integrated the following tests into the *PyNN.hardware* framework[1]:

- the *link test* to verify the connection to and from the *Spikey* chip
- the *parameter RAM test* to verify the functionality of the parameter RAM
- the *event loopback test* to test event (spike) processing in the digital part of the chip

Thus, basic functionality can be asserted by automatic means before running *PyNN* scripts. In case of error, an instructive message is displayed. Thus, the need for hardware specific knowledge is reduced.

For documentation of the low-level tests see A.1.3 (p. 44).

4.1.1 Link Test

The verification of the physical links to and from the *Facets Stage 1 Hardware* chip is the most basic test for digital functionality. The physical layer (cf. fig. 4.2) of the interface consists of two unidirectional links per direction, each transporting 8 bits of data and one frame bit. To verify the functionality of the physical links to and from the chip, a special *Bypass Operation* mode may be used. In this mode, the chip acts like a shift register, pushing received data into a FIFO queue. The same queue is used as source for output data. On the software side a testmode has been developed by Dr. A. Grübl [*Grübl*, 2007, p. 87]. After setting the control interface testmode via `SpikeNet::setChip()`[2], all input data (called *data out*) is forwarded to the output buses (called *data in*[3]). The principle of operation is shown in figure 4.2. The link test fails if input data and output data differ[4]. To test different patterns,

[1] Instructions on how to start the tests from within a Python scope can be found in the *Softies' Trac* (see section A.1).

[2] A special chip input pin called `CI_MODE` is pulled to high. This activates the *Bypass mode*

[3] Variable denomination has been done from an FPGA point of view.

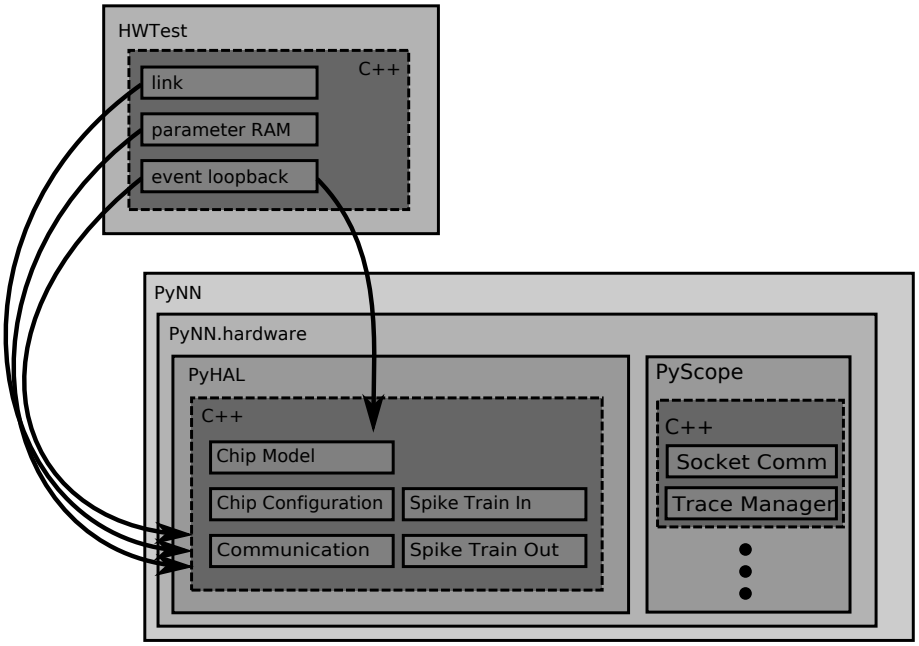


Figure 4.1: Hardware test and hierarchy of software abstraction layers

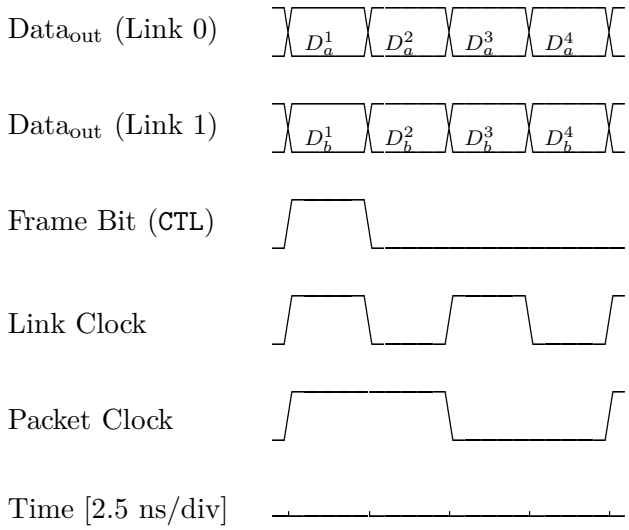


Figure 4.2: Operation principle of the link test: Bit time 0 (D^1) starts at a rising edge of the link clock, whereas bit time 1 (D^2) starts at the following falling edge. As a result, 16 data bits are transmitted within a link clock cycle. Thus, both links combined yield 64 bits of data per packet clock cycle.

a random number generator creates input data in a loop (10^3 runs by default).

4.1.2 Parameter RAM Test

The values of the various model parameters are stored within current memories, that are refreshed periodically. A controller addresses each memory cell (with address `cadr`) in a programmable sequence and a DAC writes the `value` to the corresponding cell. The write timing parameters are:

- number of clock cycles to activate boost write mode (4 bit)[5]
- number of clock cycles needed to activate normal write mode (4 bit)
- optional automatic increment of the physical write address (8 bit) with default value 1
- optional step size (4 bit) with default value 1

These parameters are stored within a look-up table containing 16 different possible setups. Along with this 4 bit look-up table address, the physical address (12 bit) and the value applied to the DAC (10 bit) are stored in the *parameter RAM*.

The utilized chip versions (*Spikey* 2 and 3) make use of almost 3000 parameters.

To test the basic functionality of the parameter RAM, a *testmode* has been developed by Dr. A. Grübl. This *testmode* writes random (but legal) values to the *parameter RAM*. After filling up the RAM, it is readout again. If the readout differs from the input, the test fails.

4.1.3 Event Loopback Test

The purpose of the event loopback test is to verify the functionality of the event (i.e. spikes coming from outside the chip and spikes going off-chip) processing modules in combination with the synchronization of the chip. In principal, the event loopback works as pipeline register. Events from the input links pass through input *event buffers*, then bypass the analog part, pass through output *event buffers*, finally arriving at the output links [*Grübl*, 2007, p. 99]. A setup to compare a random set of input spike events with the output spikes received from the chip is given by the event loopback testmode, which has been developed by Dr. A. Grübl. If the test succeeds, this indicates that the digital part of the chip is working.

4.1.4 Conclusion

The author worked through the source code, documenting it and adding meaningful error messages. With the help of *Boost.Python*, these three tests were integrated into the *PyNN.hardware* software framework. To further improve the automatic hardware tests, an analog test is under development.

[4] The *delaylines* [*Grübl*, 2007, p. 89] must be calibrated before.

[5] The output of the DAC drives a large capacitive load, as it is connected to all current memory cells in parallel. Thus, the *RC* time constant at the output is quite large and the output settling time becomes large, too. To solve this problem, a *boost mode* is implemented. The target memory cell and nine internal dummy cells are connected in parallel. The output current is increased tenfold, so the current to the target cell remains the same, but the settling time is reduced. The drawback is the decreased resolution (10 bit in normal mode), as the target to dummy cell ratio is different for every target cell.

4.2 Characterization Of The Chip

4.2.1 Noise

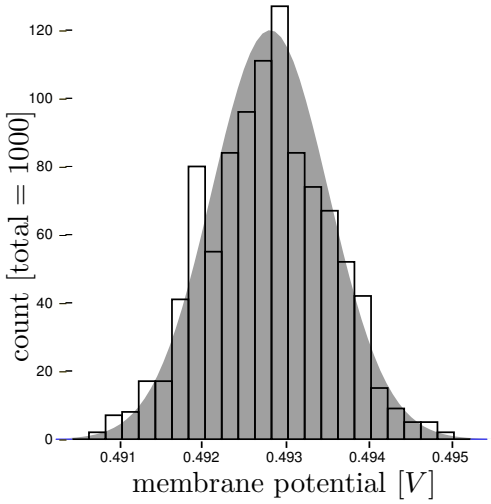


Figure 4.3: Distribution of the resting potential. A gaussian function is fitted to the data. $\mu = 0.493$ mV, $\sigma = 0.995$ mV

Operating a mixed-signal VLSI device always involves dealing with electronic noise. Several physical sources exist – like thermal (or Johnson-) noise [*Dally and Poulton*, 1998, p. 267–268] caused by thermal agitation of charge carriers or shot noise which is caused by the fact that the current is carried by discrete charges. This type of noise is typically categorized by its relation between power density and frequency. E.g. white noise is characterized by a constant spectral density. Another important type is *pink* noise with $1/f$. Noise caused by an ongoing process is called deterministic noise, as its occurrence is generated by some circuit. For example crosstalk is typically caused by capacitive, inductive or conductive coupling from one circuit to another. While investigating the *FACETS Stage 1* hardware analog behavior, noise is a common effect. Figure 4.4 shows the resting potential of a *Spikey 3* chip and its power spectrum (orange in the upper half). Multiple peaks can be identified: 330 kHz which can be traced back to the power supply and multiples of 100 MHz – which is the clock frequency of the digital part of Spikey.

Figure 4.5a shows another *Spikey 3* resting potential while a parameter update of so-called *membrane voltage output buffer* bias currents occurs. Nine digital *spikes* can be seen through crosstalk from the update circuits. As the spiking behavior is not affected, it is assumed to be limited to the readout chain. A histogram of the resting potential distribution can be seen in figure 4.3. The fit of a gaussian function to the data yields a σ of ca. 1 mV. The contribution from the readout chain is unclear.

While calibrating the synapse drivers a method called STA[6] is used. This method extracts analog membrane traces triggered by output spikes. After averaging over several (up to 10^5) samples the gaussian noise is small enough to unveil membrane structures as small as the typical noise on the analog output pin. In figure 4.5b an EPSP is shown. A digital *spike*[7] can be seen just before the EPSP starts. This is caused by crosstalk from digital circuits upon the arrival of the digital information of the spike.

4.2.2 Neuron Readout

First tests of *Spikey 2*[8] indicated spike readout problems. As soon as two neurons within certain subsets of a readout group[9] are recorded, the readout is prone to deadlock after a short time. The deadlock seems to occur most likely when two neurons fire in quick succession. Thus, a typical setup locks after only a handful of output spikes (see figure 4.6). If multiple

[6] Spike-Triggered Averaging
[7] In this case an action potential is **not** meant.
[8] FACETS Stage 1 hardware chip version 2
[9] 64 neurons are grouped together. Three groups form a block.

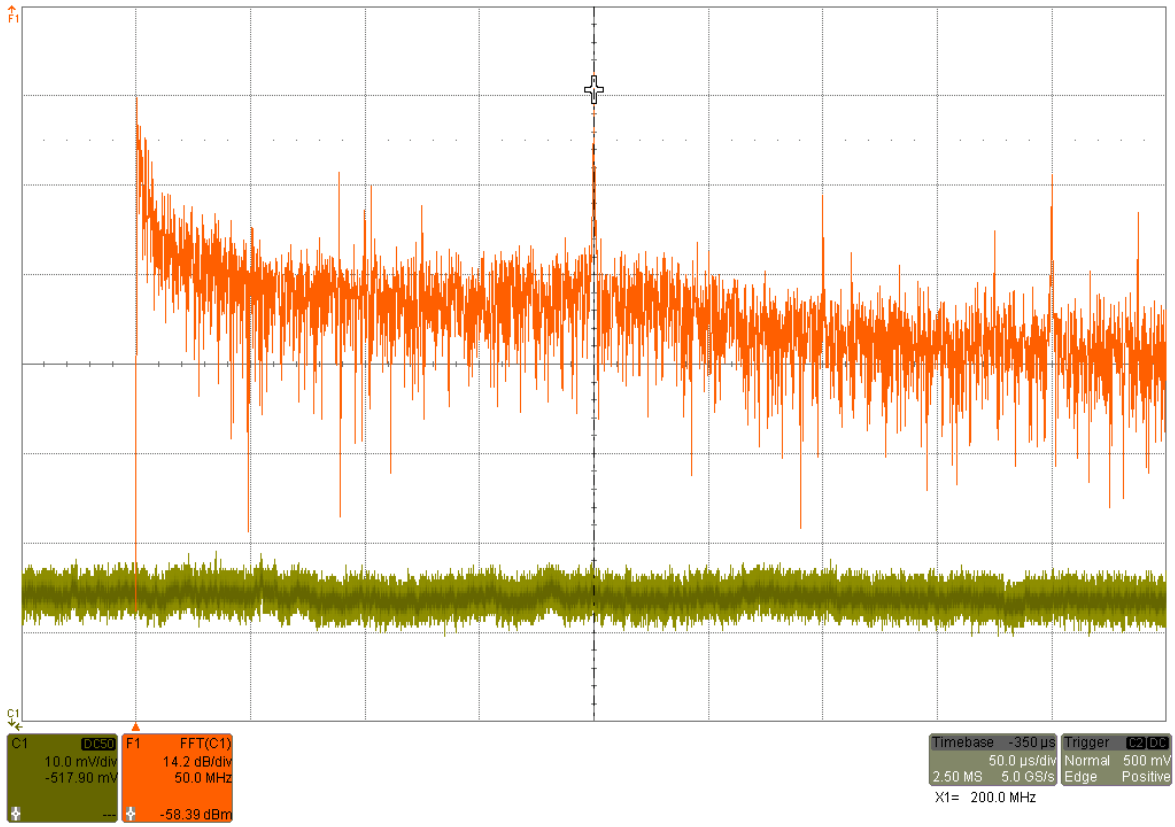
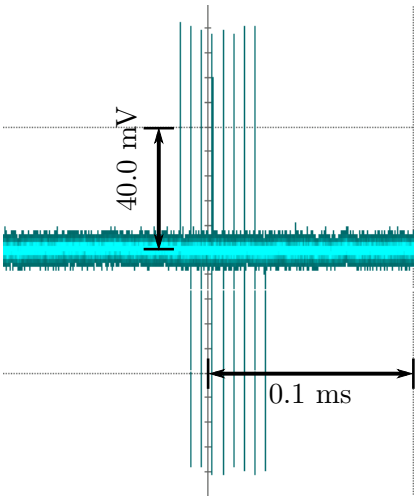
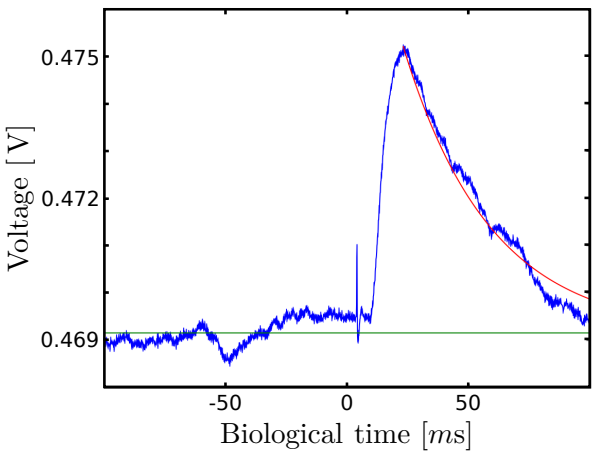


Figure 4.4: Resting potential (Channel 1, green) and its power spectrum (orange). Peaks can be seen at multiples of 100 MHz (centered on 200 MHz, 50 MHz/div).



(a) Crosstalk on read out (resting) membrane potential due to parameter updates to the nine *membrane voltage output buffer* bias currents.



(b) A digital *spike* ($t \approx 5\,\text{ms}$) can be seen before the EPSP (spike-triggered averaged over 50000 runs)

Figure 4.5: Crosstalk on *Spikey 3*

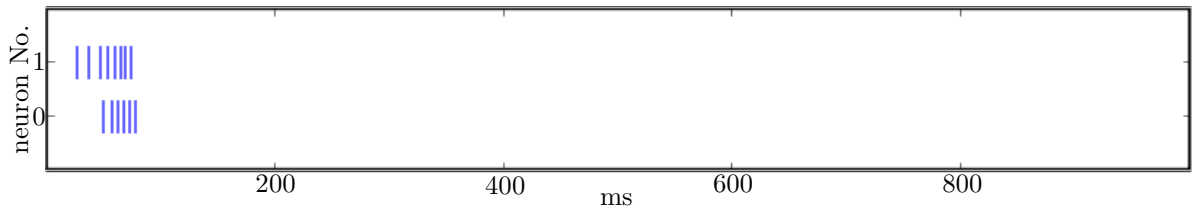


Figure 4.6: Rasterplot showing two recorded neurons. Artificial setup to sustain maximum fire rate over experiment runtime (1000 ms bio. time). After 100 ms bio. time, readout enters a deadlock state.

neuron readout is not required, a workaround is possible: to read out every neuron, the experiment is repeated for each neuron recorded. This is a viable solution if single neuron statistics suffice or if the spike output is highly correlated between multiple runs of the same experimental setup. Another approach is to map neurons that are to be recorded to different readout groups. Thus, up to six[10] neurons can be read out in parallel. During the process of high-level testing, the author discovered that on most chips up to 9 neurons per group can be read out. The exact number depends on the individual chip[11]. Five out of six tested Spikey 2 chips, are able to record an arbitrary neuron and the last 8 neurons within a group of 64 neurons. Thus, up to

$$3\ \frac{\text{groups}}{\text{block}} \cdot 9\ \frac{\text{neurons}}{\text{group}} \cdot 2\ \text{blocks} = 54\ \text{neurons}$$

can be recorded in parallel. In the process of pre-production testing for chip version 3, a possible routing bug that generated a high *RC* time constant on some circuits was suspected of being responsible for the readout bug. Unfortunately, after testing the finally produced chip version 3, the problem remains the same. Further analysis of the chip layout is necessary to hunt down the bug.

A workaround for this problem – developed by the author – is presented in section 4.3.2.

4.2.3 Parameter Stability

A whole set of programmable voltage parameters within the first two versions of the *FACETS Stage 1* hardware show a significant voltage drift. This drift – especially on threshold parameters – causes an unstable spiking behavior as firing rates under identical settings change within an experimental run. Additionally, as the periodical update process of most essential parameters on the chip runs asynchronously with respect to the start of an experiment, identical setups yield different behavior. This introduces a systematic error on all measurements (cf. figure 4.7). Typical durations for experiments are $1 \ldots 20\,\text{s}$ (biological time) which corresponds to $10 \ldots 200\,\mu\text{s}$ real-time. As can be seen in figure 4.7, the drift shows approximately linear dependency on time. The typical refresh period is 2 ms, which results in ca. $13\,m\text{V}$ peak-to-peak voltage drift. This corresponds to $5\% \ldots 10\%$ of the dynamic range. Other voltage parameters were tested and the drift was confirmed. The parameter drift in comparison to noise is ca. 10 times larger – more details can be found in section 4.2.1

A workaround for this problem – developed by the author – is presented in section 4.3.3.

[10] 384 neurons in 6 groups comprising 64 neurons each

[11] This implies an *RC* timing problem, as different chips from different wafers yield different deadlock characteristics.

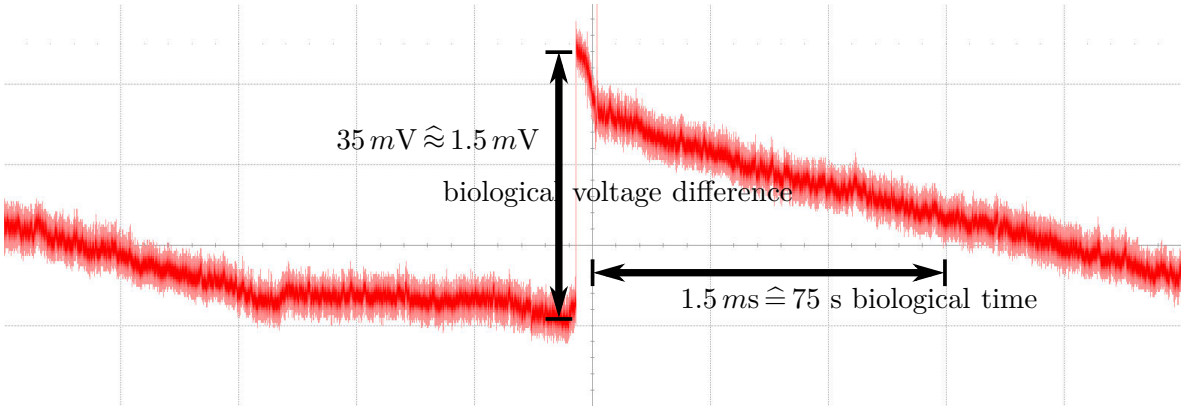


Figure 4.7: Drift dynamics of a membrane potential in rest. Second version of the *Stage 1* hardware. The resting potential parameter is periodically refreshed, but during one period ($T = 2$ ms up to 8 ms) its value is decreased by parasitic leakage currents.

4.2.4 Spike Timing Dependent Plasticity

To test the STDP functionality (see 2.3.2 for a description) of the *FACETS Stage 1* hardware, the following measurement method was developed by Dr. A. Grübl, D. Brüderle and the author (a schematic of the experimental setup is given in figure 4.8a).

Within each (excitatory) synapse (as described in section 2.3.2), two variables F_c and F_a are accumulated to $\sum F_c$ and $\sum F_a$ respectively. To measure correlations, the only way to acquire any correlation data is to read out the correlation flags which indicate if $|\sum F_c - \sum F_a|$ is significant[12] and whether $\sum F_c$ was greater (*causal*) or smaller than $\sum F_a$ (*acausal*). To generate correlations, a pre-synaptic spike sent into an *observed synapse* has to arrive at a spiking neuron. Now, the correlation measurement takes place, and if the correlation was *large* enough, the corresponding flag is set. Typically (as in biological measurements) multiple correlated spikes are necessary for $\sum F_a$ (or F_c) to be sufficiently large to raise the correlation flag.

The idea is to decouple pre-synaptic input spikes (of the observed synapse) from post-synaptic firing, thus avoiding a causal correlation: strong synapses trigger post-synaptic firing. A single weak[13] synapse called *observed synapse* is used for correlation measurement by means of sending a pre-synaptic spike with an offset in time Δt with respect to the pre-synaptic spike of the trigger synapses. If Δt is negative, a causal correlation between observed synapse and post-synaptic firing is created and vice versa. Figure 4.8a shows a schematic of the measuring method.

To trigger a single output spike at a fixed time, various constraints of the hardware have to be considered: first, only one simultaneous input spike is possible per synapse driver group, thus only one out of 64 synapse drivers may be used. Second, as a single synapse driver typically is not strong enough to initiate a post-synaptic firing, multiple strong synapses have to be used (*trigger synapses*). The weight of the *observed synapse* is set to minimum (max-min ratio 15 : 1) thus minimizing its effect on the post-synaptic neuron. As 256 synapse drivers split up into four groups, three trigger synapses with maximum weight plus one observed

[12] This threshold is a configuration parameter.
[13] Therefore, this synapse shall not influence the neuron membrane potential.

synapse with minimum weight are created.

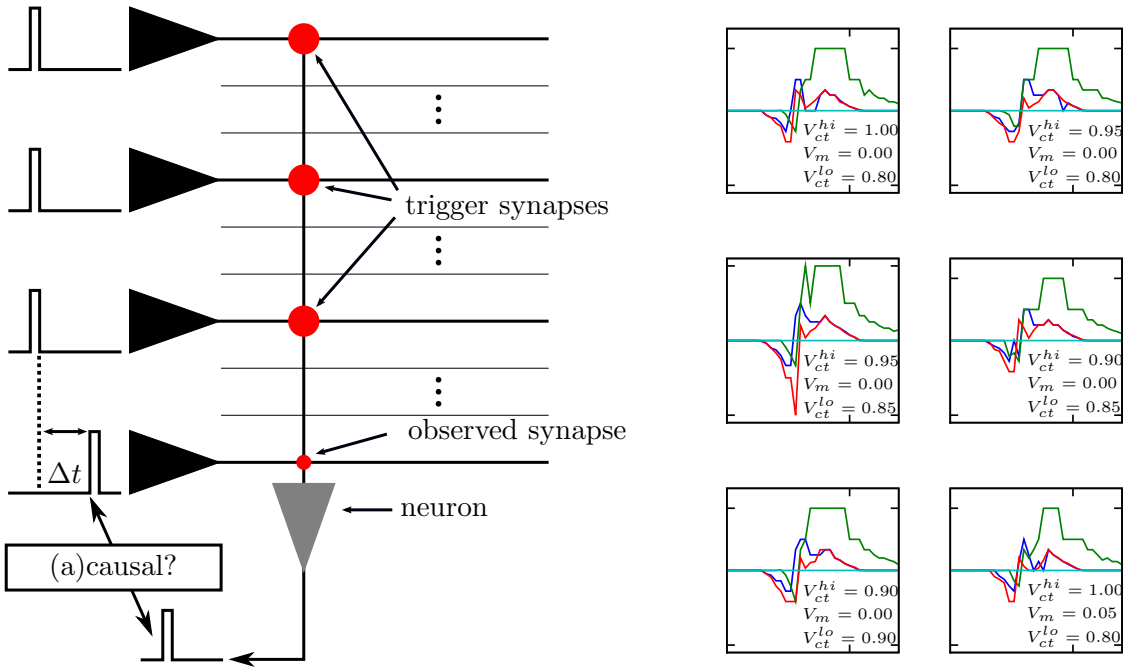
A spike train consisting of equidistant spikes is generated. This spike train is transmitted to all trigger synapses and to the observable synapse with an offset in time Δt. The post-synaptic neuron receives three strong EPSPs and one weak EPSP of the observed synapse. Subsequently, the neuron fires a single output spike and the correlation flag of the observed synapse is read out. Typically, a single correlation measurement is insufficient. Thus, the process is repeated with multiple equidistant spikes within the input spike train until the correlation flag indicates a correlation. The number n of input spikes needed is recorded. It is inversely proportionally to $F_c(\Delta t)$[14]. If an arbitrary limit of the number of spikes is reached, $F(\Delta t)$ is defined as zero. Therefore, defect synapses reporting no correlations are handled in a deterministic way.

Thus, starting with a negative value Δt_{min}, step size t_{step} and Δt_{max} the function $F(\Delta t)$ (eq. 2.4) is scanned. To discover a valid configuration of the hardware, multiple parameters controlling the amplitude of $F_c(t)$, the amplitude of $F_a(t)$ and the significance threshold[15] (eq. 2.6) have to be sweeped. However, as most hardware neurons do not fire reliable enough when triggered by only three pre-synaptic input spikes a weight calibration routine is needed. The weights of the trigger synapses are calibrated to trigger exactly one post-synaptic spike. Further improvements accelerate the measurement: first, to detect if correlations can be found at all for a given Δt, the routine starts with a high number of correlated spike pairs – if no correlation flag is set, $F(\Delta t)$ is set to zero. Second, a multi-neuron measurement is implemented. Therefore, multiple synapses connected to different neurons can be tested in parallel.

Figure 4.8b shows $F(\Delta t)$ for multiple neurons ($\Delta t = -25 \ldots 10\,ms$) and different hardware parameters. The resulting functions resemble roughly the implemented model (see fig. 2.4 on page 9) and on all three tested chips, synapses can be found which work correctly. However, many synapses show erratic behavior signaling unstable correlation flags in consecutive (identical) runs. To find multiple synapses on a single chip working equally well with the same set of parameters has not been possible yet. Only possible errors in low-level code remain as possible reasons for this problem beyond a fundamental misbehavior of the hardware itself.

[14] Δt includes a hardware specific offset in time – which is a configurable *spike delay*. Thus, the resulting $F(\Delta t)$ is shifted in time.

[15] In hardware two values are used to allow for both, a significance offset V_{ct}^{lo} and a significance difference $V_{ct}^{hi} - V_{ct}^{lo}$ between causal and acausal correlations.



(a) Measuring principle: multiple synapses get the same pre-synaptic input and trigger a post-synaptic spike. A single *observed* synapse transmits a time shifted spike train to induce a time difference between its pre-synaptic input and the post-synaptic firing of the neuron.

(b) A clipped view of an STDP parameter sweep. Three different colors represent three synapses.

Figure 4.8: Measuring STDP curves: setup (*left*) and extract of a parameter sweep (*right*)

4.3 Workarounds for Chip Imperfections

4.3.1 Summary Of Main Problems

In previous sections, various problems in operating the *Spikey* chip were described:

- *Neuron Readout*: if two or more neurons are being recorded, a deadlock may occur.
- *Parameter Drift*: on *Spikey 2* essential voltage parameters show a significant drift, thus adding a systematic error on most measured variables.
- *Long-term plasticity*: if multiple synapses are to be used, STDP[16] cannot be set up to work for all.

For the first two problems, workarounds were developed by the author which are described in the following sections. In addition, the existing routine for calibration of voltages generated on-chip was improved. The calibration mechanism is explained in section 4.3.4.

4.3.2 Neuron Resets

As demonstrated in section 4.2.2, a deadlock can be triggered by recording more than two neurons. More neurons can be recorded, if one uses the eight neurons at the end of a neuron group. But a comprehensive workaround for recording is preferable. Dr. A. Grübl and the author tested and developed a software solution that uses so-called *neuron resets* to clear the lock. A neuron reset disables the spiking mechanism and resets the so-called *priority encoders* which handle, among other things, event (spike) delivery. The analog membrane time course is not affect – integration continues. However, to trigger a spike, the membrane potential has to stay above threshold until the reset is over. These three effects – lost spikes, delayed spikes and distortion of the membrane potential – may significantly change the dynamics of a experimental setup. A crosscheck verifying the dynamics of representative neurons without this workaround is essential. This has to be done for every unchecked experimental setup.

As a first test, a single neuron reset was triggered after different time intervals (relative to the start of the experiment). It turned out that a single neuron reset is insufficient: a reset neuron remains prone to deadlock. In the following, different stages of the workaround are described. Figure 4.9 illustrates the insertion of neuron resets. The membrane potential recovers toward resting potential.

A reset every nth Input Spike

As a first approach, a straightforward implementation was tested. The function that sends a spike train was modified to insert a neuron reset after every nth spike. However, as the low-level packet generation algorithm lies below this layer, detailed control cannot be guaranteed. The package generation algorithm tries to maximize the bandwidth to the Spikey chip. This is achieved by sending spikes as early[17] as possible to the chip. As the exact timing difference between chip and control program is not accessible by higher-level functions, this solution is not universally suited. However, because of its simplicity, it was implemented for testing purposes. First tests showed that the deadlocks could be released. But to obtain a solution that is robust with regard to varying input/output rate ratios, a more controllable solution

[16] Spike timing dependent plasticity
[17] typically 128 clock cycles ahead

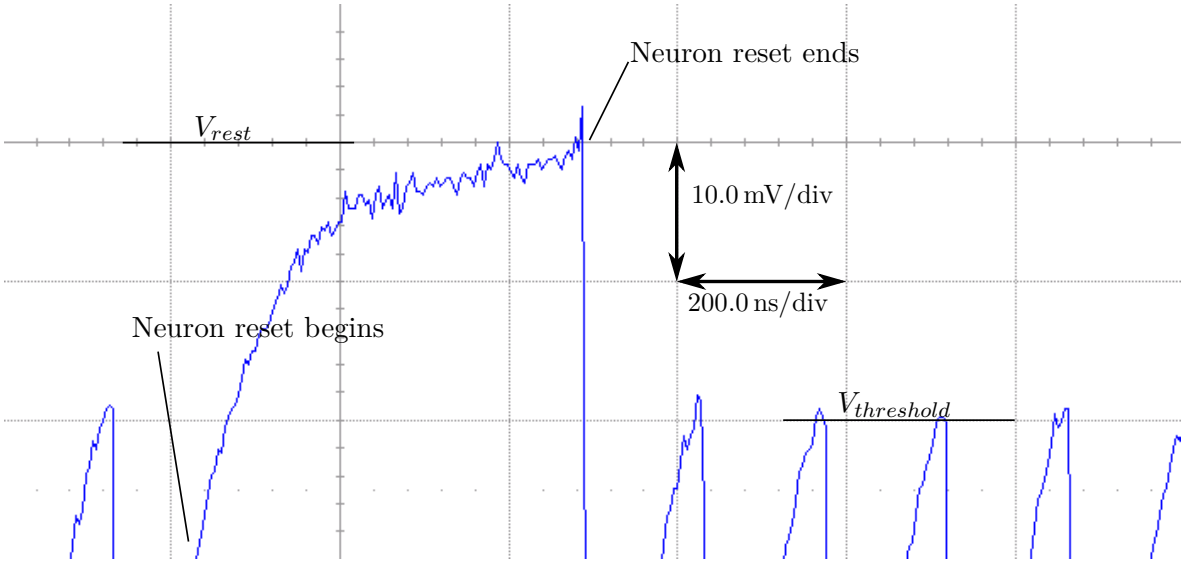


Figure 4.9: Artificial setup ($V_{rest} > V_{threshold}$, therefore the neuron fires at a high rate; reset duration set to a large value for demonstration purposes) to illustrate the insertion of neuron resets. As the spike mechanism is disabled during neuron resets, the membrane potential recovers towards V_{rest}. To zoom in on the interesting part of the membrane trace, the majority of the trace is below the plotted y range.

is needed. A solution that allows for a distinct control of reset density independent of input spike train density is crucial for recurrent networks or sparse input. Figure 4.10 shows a rasterplot of two neurons (cf. section 4.6) being reset multiple times.

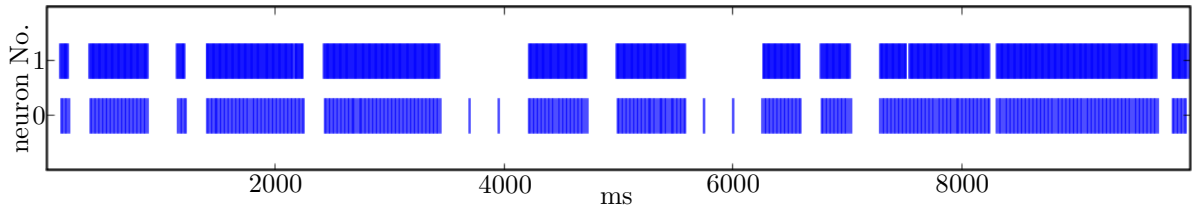


Figure 4.10: Rasterplot showing two recorded neurons. Artificial setup to sustain maximum fire rate over experiment runtime (10000 ms bio. time). Readout locks multiple times, with neuron resets (shorter than 1 ms bio. time) in between. As the high output rate increases the deadlock probability, a higher reset rate should have been chosen.

Fixed Rate

The second approach alters the low-level event packing algorithm. This algorithm generates the program[18] that is played back by the FPGA (for a detailed description see *Grübl* [2007, p. 136]). To avoid lost input spikes, a neuron reset may only be inserted after completed

[18] The so-called playback memory program. Typically a vector of spike trains is passed to function `SC_Sctrl::pbEvt`, which generates the program. At this level the system's *real* time, including event time stamps, is known.

event commands (i.e. input spikes) and if the following event is late enough[19]. A variable[20] controlling the distance between two neuron resets is defined. Thus, the neuron resets are inserted at a fixed rate if no spikes are to be delivered. Figure 4.11 shows the membrane

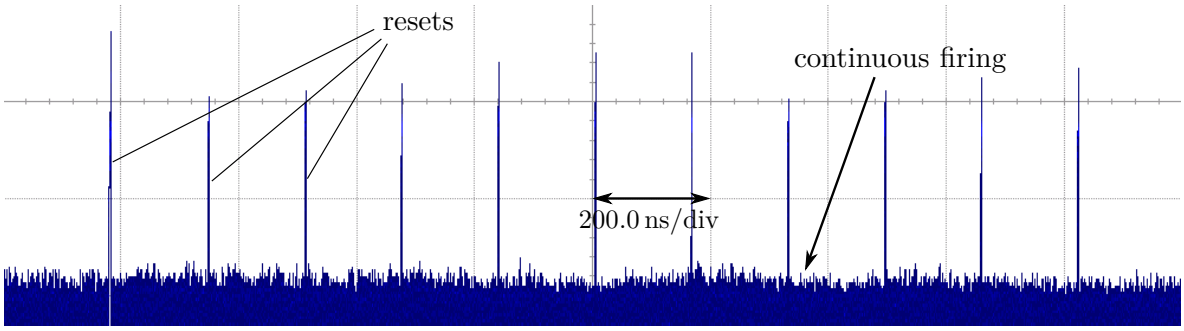


Figure 4.11: Artificial setup to trigger continuous firing (cf. fig. 4.9). The membrane potential recovers to resting potential if the neuron reset is held. The resets are inserted at a fixed rate.

potential of an artificial setup (neurons firing continuously) to demonstrate the insertion of neuron resets. As the neuron reset disables the spike mechanism the neuron recovers towards the rest potential as long as the reset is held.

Poisson Distributed

As neuron resets disable the spike generation mechanism, the neuron recovers towards resting potential. Thus, a change in neuron dynamics occurs. To avoid possible artifacts, another sophistication of the reset insertion is needed: the poisson distributed reset insertion. Like poisson sources in neural networks, a poisson process generates a number n with a given number of expected occurrences λ. Then, n numbers are picked out of a flat (uniform) distribution. Numbers thus obtained are sorted and inserted into a vector. The entries within this vector are spread to the experimental runtime. Now the vector contains times, at which neuron resets are inserted if the previously mentioned distances to real events (spikes) are large enough. Figure 4.12 shows an artificial setup to demonstrate the insertion of resets.

Conclusion

Figure 4.13 shows the measured output rate of an arbitrary neuron plotted against the mean distance between two neuron resets. In this setup, no input spikes were sent to avoid dropping of resets. To force continuous firing, the resting potential was set above threshold voltage. For a reset duration ≥ 2 bus clock cycles, the output rate is significantly higher than zero[21]. Inserting neuron resets with a mean distance of 64 maximizes the output rate. If only one neuron is recorded, the output rate is about 800 Hz (in biological time). However, while analyzing the analog membrane dynamics, a bug was found: some input spikes are delayed by 512 cycles. As this bug does not occur without the neuron reset workaround and the length of delay suggests a wraparound of the 8 bits internal *Spikey* time, it is assumed to be located within the event packing algorithm where the timing is calculated. It is for this reason

[19] A neuron reset takes time – the reset duration plus the time for reset deactivation.
[20] `resetdist`, defined in bus clock cycles
[21] Without neuron resets, the mean output is close to 0 as typically just one spike is recorded.

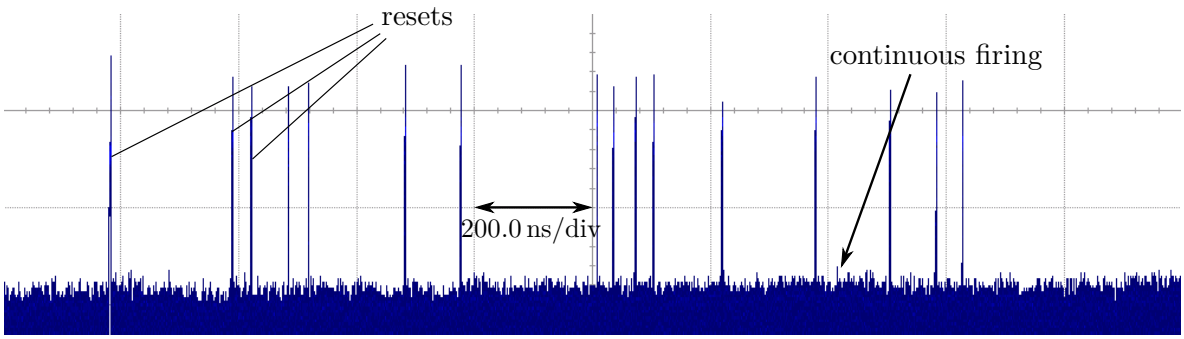


Figure 4.12: Artificial setup to trigger continuous firing (cf. fig. 4.9). The membrane potential recovers to resting potential if the neuron reset is held. Neuron resets are inserted in a random fashion.

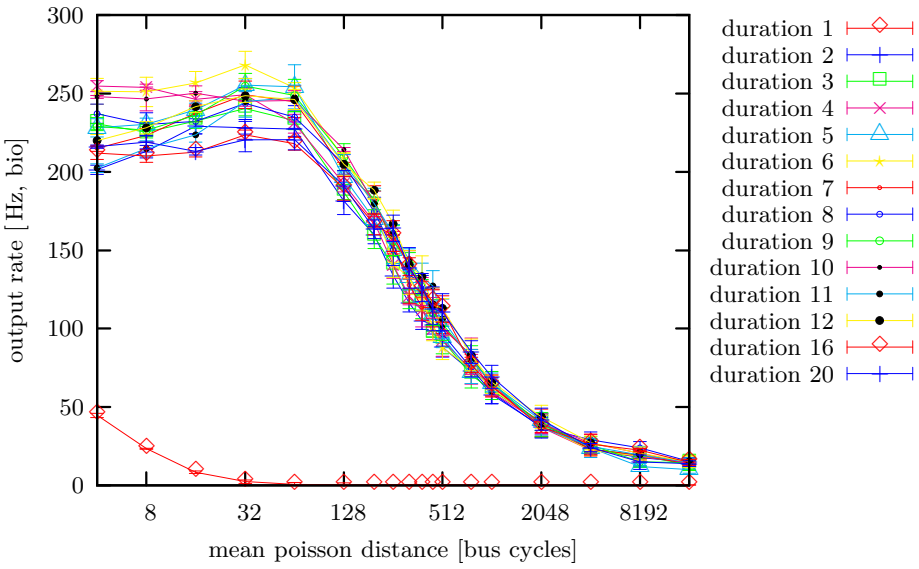


Figure 4.13: Output firing rate is plotted against the mean distance of neuron resets. Different lines mark different resets durations. A single recorded neuron fires at about 800 Hz (bio. time).

that this promising workaround is not usable yet. Low-level code analysis did not reveal an obvious bug. Therefore, a low-level system simulation is needed to further investigate this case. It is not part of the author's field of activity, the responsible group is occupied by other work.

4.3.3 Parameter RAM Update

As introduced in section 4.2.3, *FACETS Stage 1* hardware version 2 shows a voltage drift on a whole set of programmable voltage parameters. To deal with this effect, a combined hardware and software solution was developed. For different parameters and write sequences the optimal write timing was measured. An implementation in pseudocode is given in algorithm 1 and in the following paragraph:

The parameter RAM periodically updates the current memory cells. A global refresh time can be defined by summing up the write time for all addressed cells. To decrease the global refresh time and thus minimize the drift effect, an optimal write timing has to be determined. As the preceding value written by the DAC influences the following write process[22] the measurement has to take into account combinations of two values: a precurser and a successor value

Most parameters on the chip are built identically as current cells. While these parameters have probably an insignificant drift with respect to other sources of noise [*Schemmel*, 2008] (cf. section 4.2.1), they form the vast majority[23] of all parameters.

Furthermore, the measurement of most of these cells is not possible directly. Therefore an indirect measurement method has been developed by the author. For the current cells the parameter *drviout* [*Grübl*, 2007, p. 177] can be measured indirectly. *Drviout* controls the amplitude of the conductance time course generated by a pre-synaptic spike. Thus, the measured amplitude of a conductance pulse is related to the parameter *drviout*.

For a given target value and increasing write time, the parameter should converge[24] towards a saturation value. Therefore, while increasing the write time, the height of a EPSP[25] saturates as the DAC output converges. Thus, the best timing for a given target value $drviout_{target}$ immediately succeeding another value $drviout_{precursor}$ can be identified by measuring all write configurations (consisting of write mode (normal or boost), write time (2^N with $1 \leq N \leq 16$), target, and its preceding value). These measurements were performed manually, a plot of the data is shown in figure 4.14 (measuring points are green). The estimated optimal write times have been exported into a C++ matrix.

The parameters are sorted by ascending value, as high-to-low writes are slower (yellow in figure 4.14). For each parameter the optimal timing can be determined by accessing the matrix (x_1 = preceeding value, x_2 = target value). This procedure yields a good write time while minimizing the global refresh time. The implemented code accelerates the global parameter update rate by a factor of approximately 4. As the drift shows a nearly linear dependency on time (cf. fig 4.7), the corresponding parameter drift is reduced by a factor of 4 as well.

[22] The DAC writes values sequentally. So for each value a preceding value exists. The output of the DAC drives a relatively large capacitance, therefore different preceding values – as well as address blocks – change the writing chacteristics.

[23] Current cells outnumber parameter voltage generators by a factor of 50. Thus the timing of these cells is crucial to the global refresh time. The parameter voltages can be multiplexed to an analog output pin and are easy to measure.

[24] After a sufficiently long write time, the DAC output settles at a value.

[25] Excitatory Post-Synaptic Potential

```
1:  procedure setValue(pos, value)
2:      write value at pos          ▷ manipulating playback memory program at position pos
3:  end procedure
4:
5:  procedure setTiming(pos, n)
6:      set new write timing at pos to 2^n          ▷ for playback memory position pos
7:  end procedure
8:
9:  function getPos(s)                                  ▷ s: synapse driver index
    return pos                                          ▷ pos: position of s
10: end function
11:
12: for s in considered drviout parameters do
13:     pos ← getPos(s)
14:
15:     for f ← 0 to 1023, step ← 128 do
16:         setValue(pos-1, f)
17:
18:         for d ← 0 to 1023, step ← 128 do
19:             setValue(pos, d)
20:             n ← 1
21:
22:             while height of EPSP changes do
23:                 setTiming(pos, n)
24:                 n ← n + 1
25:                 if n > 15 then
26:                     break                       ▷ EPSP did not saturate for s
27:                                                 ▷ predecessor f, target d
28:                 end if
29:             end while                   ▷ 2^n is minimum required write timing
30:         end for
31:     end for
32: end for
```

Algorithm 1: Pseudocode describing measurement of write timings.

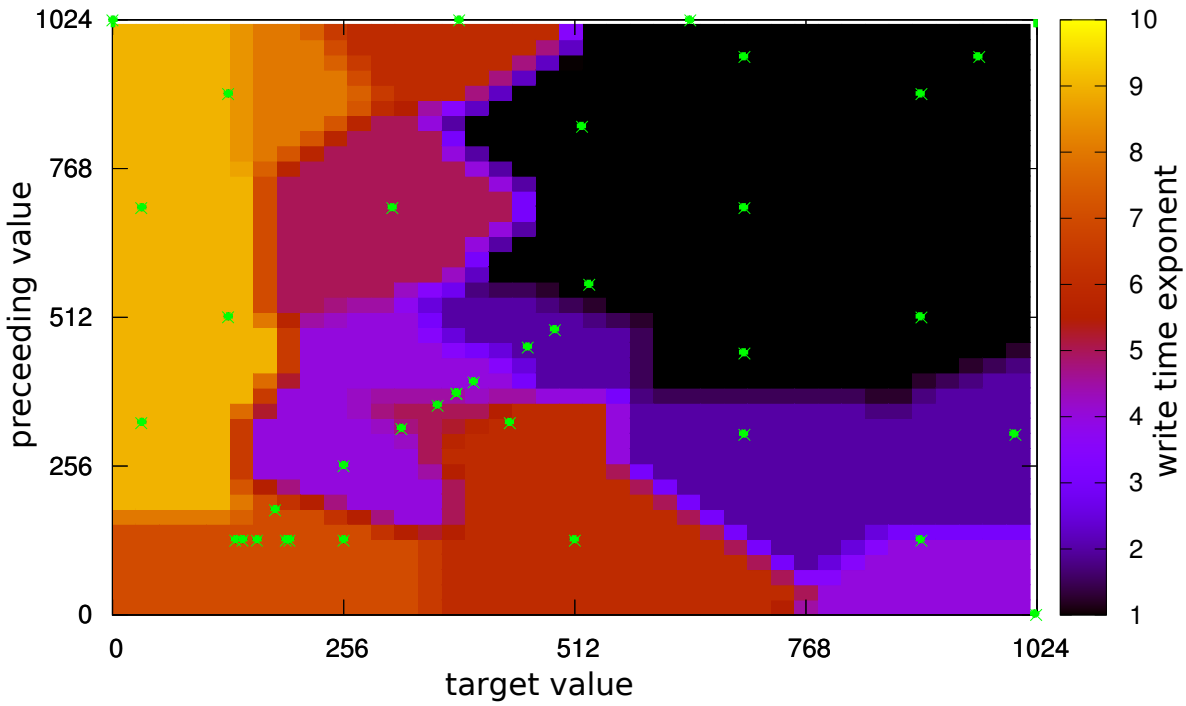


Figure 4.14: Optimal write timing (for current cells on FACETS Stage 1 hardware version 2) depending on value pair (preceding and target value) plotted. DAC write time is given in $2^{n=1\cdots10}$ cycles. The measured points are green, the underlying grid is generated by next-neighbor interpolation. Especially small values are hard to reach (yellow color).

As the large drift only affects the 46 voltage parameters, in a final version the corresponding parameters were written three times per refresh cycle, thus effectively increasing their refresh rate by an additional factor of 3. It is for this reason that the effect of parameter drift on essential model voltages was reduced by a factor of 12. The reduced drift was in the same order as the system inherent noise (cf. section 4.2.1).

A third version of the *Stage 1* hardware was submitted in late 2007, redesigned to fix that bug such that no workaround would be necessary at all anymore. The first run which was delivered by the producing company was erroneous due to a mistake during production. The company found their mistake and so the second trial sent in late April 2008 was functional. Tests show that the previously seen drift has vanished due to a better parameter storage design (see Figure 4.15).

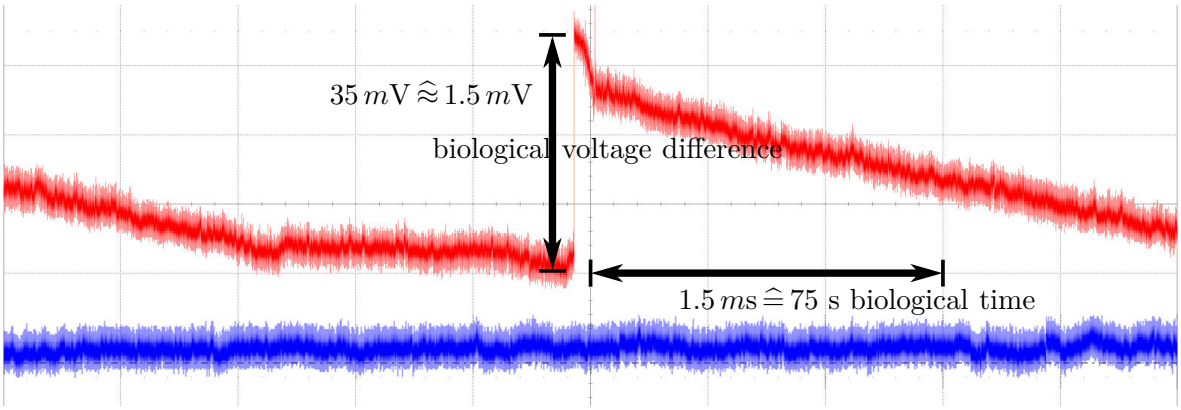


Figure 4.15: Drift dynamics of a membrane potential in rest. The resting potential parameter is periodically refreshed, but during one period ($T = 2$ ms up to 8 ms) its value is decreased by parasitic leakage currents. Red: Second version of the Stage 1 hardware (*Spikey 2*) Blue: Third version of the Stage 1 hardware (*Spikey 3*). The drift has vanished.

4.3.4 Parameter Voltage Generator Calibration

Several model parameters (e.g. reversal potentials, threshold voltage, resting potential) require the supply of voltages rather than currents. On the *Spikey* chip, these voltages[26] are generated on-chip by converting an output current of a DAC to a voltage by means of a poly silicon resistor of $R = 10k\Omega$. Typical resistor mismatch reaches up to 30 %, thus a calibration is needed to ensure precise parameter values. In total, a Spikey chip contains 46 voltage parameters (so-called *vouts*). The relation of DAC value applied and measured output voltage has to be measured. Based on this curve a bijective correction function can be determined. Here, a linear function yields a good approximation of the measured curve. A previously implemented calibration routine [*Ostendorf*, 2007] dealing with this task was improved by the author to allow for more robustness with respect to hardware varieties. Instead of manually choosing V_{min} and V_{max}, a plateau detection algorithm was added and the linear fitting method was replaced by a more robust library implementation using the GSL[27]. Thus, varying plateaus and slopes can be automatically calibrated. Finally, after

[26] A few voltages are generated externally.
[27] GNU Scientific Library

calibrating all voltage parameters, a unit test to detect failed calibrations was added. Figure 4.16 shows a calibration run of *Spikey 3 #27*[28]. In the upper figure, an outlying voltage parameter (9) can be seen. The lower figure demonstrates the plateau detection and the resulting fit to the *linear part*.

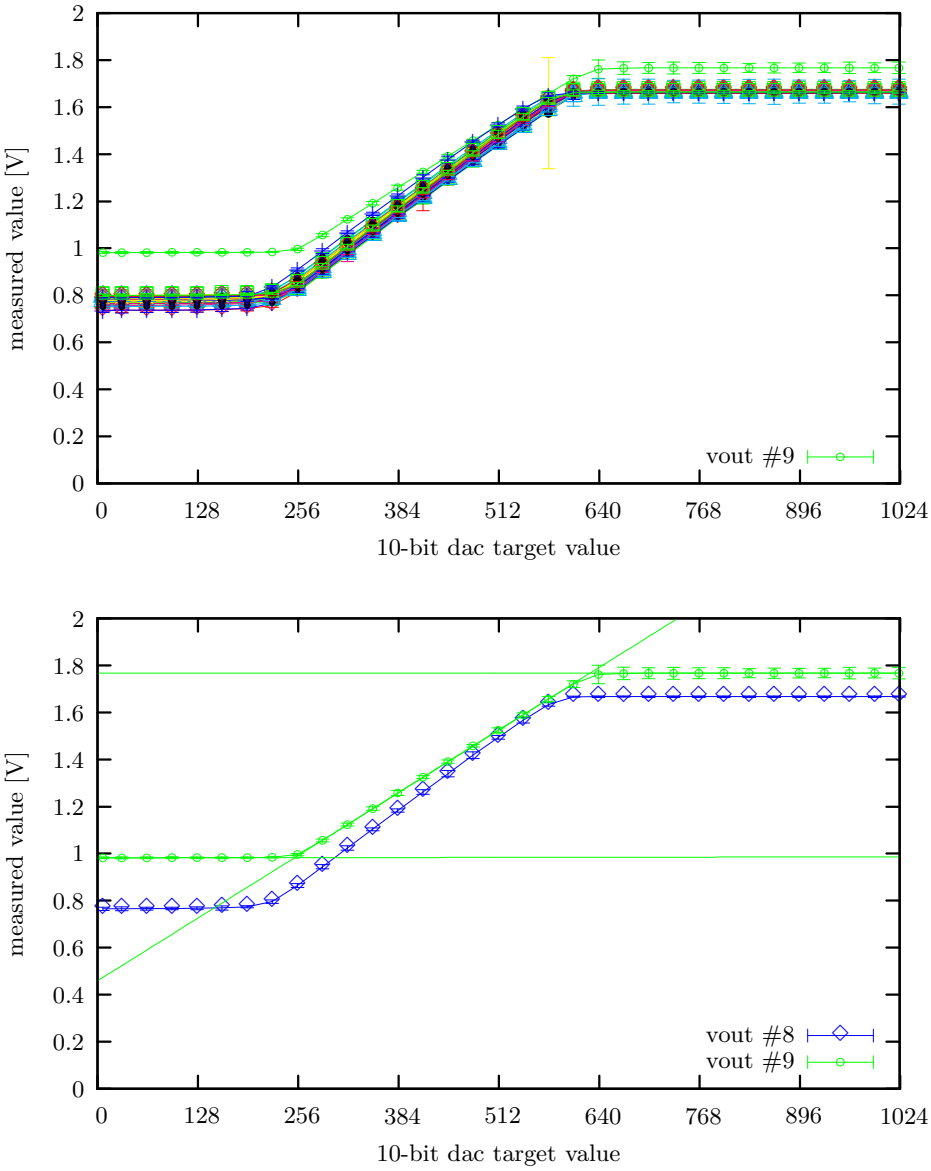


Figure 4.16: Calibration of parameter voltage generators on *Spikey 3* chip id #27. The first figure shows a measurement of the output voltage versus DAC value. Most curves are similar, whereas parameter voltage #9 is outside the common range. For calibration, the lower and higher plateau have to be found, as well as a linear function fitted in between. Measurements were extremely precise, thus for better visibility all error bars are enlarged by a factor of ten.

[28] FACETS Stage 1 hardware version 3

5 Improved Software Flow and Tools

5.1 Software Project Management

5.1.1 Build Process

The *PyNN.hardware* software framework (including low level code) consists of ca. 40,000 source lines of code (SLOC). Most of it (ca. 30,000) is C/C++ code or Python (ca. 5,000). The low level interface is written in C++ and in the course of development frequent recompilations (`make clean; make`) are needed, hence short build cycles are desired. Some small modifications to eight major *make files*[1] yielded a speedup of 4.2 after the first compilation (table 5.1). A prefix has been added to every C/C++ compile command. The following paragraph describes this prefix called *ccache*.

make	with ccache [s]	unoptimized [s]
first run	119.4 ± 0.2	112.7 ± 0.2
nth run	26.6 ± 0.3	112.7 ± 0.2

Table 5.1: Running *PyNN.hardware* make on a P4, 2.4 GHz, 2 GiB, Ubuntu 8.04.

Ccache is a tool to speedup recompilation of source code. It caches the output of (C/C++) compilation to avoid recompilation of the unchanged source code. To ensure that the source is unchanged, the `-E` compiler flag is used, which prints preprocessed code. In addition to this output, the command line options, the size and modification time of the compiler is hashed using MD4[2] . A first-time compilation triggers the creation of a cache file, which contains all compiler output including stdout and stderr messages. When the compilation is done a second time — using the same sources, same command line options and the same compiler — ccache is able to provide matching compiler output (including messages) from the cache.

Ccache can be used in conjunction with *distcc*, which is a tool for distributed compilation across multiple computers. Real compilation will be done using distcc. In case of a cache hit, ccache will supply the local compiler output.

5.1.2 Source Code Management

As multiple developers work together on the same source code, during the normal workflow bugs are found, missing features are identified and incompatible changes are being worked on. A tool organizing development efforts is beneficial for all involved. Therefore an application called *Trac*[3] was set up by the author for the PyNN.hardware project.

[1] Among other things, make is a utility for automatically building large applications. Make is part of POSIX [*IEEE*, 2004].

[2] message digest algorithm (implements a cryptographic hash function)

[3] `http://trac.edgewall.org/`

Trac is a web-based tool for bug tracking and project management. It also includes access to an underlying source code repository within a version control system, like Subversion. For documentation purposes, a wiki is also embedded. Bugs can be assigned to, and commented by, a developer in charge of the related source module.

At the time of writing, 37 bug reports and 32 wiki pages (mostly documenting how to get PyNN.hardware running) can be found[4].

5.2 Korescope

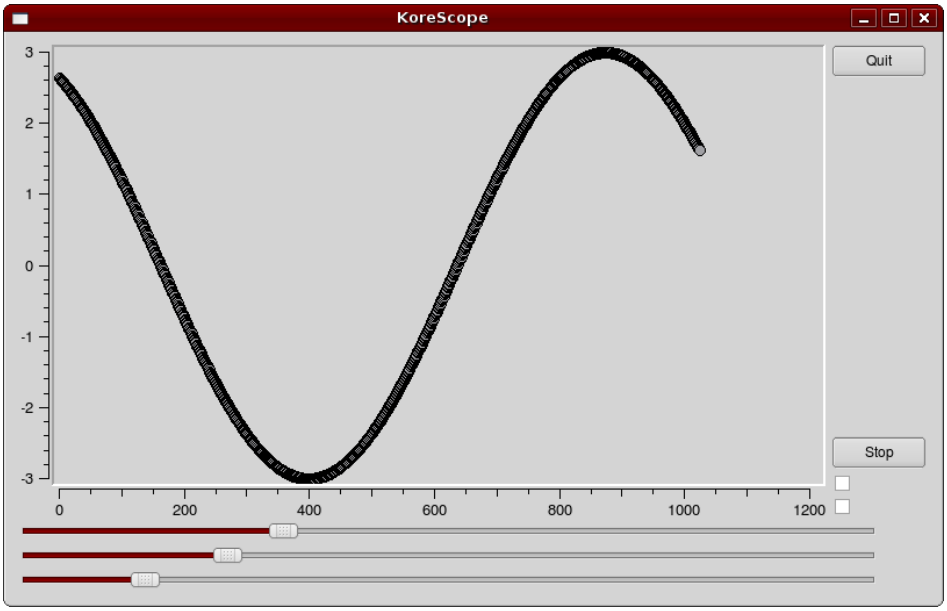


Figure 5.1: Screenshot of Korescope showing test data. The generated example plot is controlled by the sliders controlling amplitude, frequency and phase of the plotted sinus.

When operating a complex hardware device, there is often need for a graphical interface for parameter configuration. The Facets Hardware enables a low-latency response to input changes and therefore intuition-guided exploration of parameters is possible. Based on Qt 4.2 [*Trolltech*, 2006], PyQt 4 [*Riverbank Computing Limited*, 2007] and PyQwt 5 [*Colclough and Vermeulen*, 2007], Korescope is a GUI (figure 5.1 to use on top of an arbitrary PyNN [*Ensemble*, 2008] script. It is inspired by Borescope [*Bill*, 2008], that provides similar features, but requires a major restructuring of the underlying PyNN script into a fixed class hierarchy. The use of callback[5] functions minimizes otherwise necessary changes to the PyNN script. Exemplary code is shown in listing A.1.

[4] `https://cetares.kip.uni-heidelberg.de/cgi-bin/trac.cgi` (KIP-intern, LDAP login required)

[5] A callback is executable code passed to other code. This allows a low level software layer to call high level code, reducing the need for high level code changes.

5.3 Code Profiling

A high conductance state [*Destexhe et al.*, 2003; *Kumar et al.*, 2008] test developed within the group [*Kaplan*, 2008] revealed serious performance problems for long experimental run times. Using the Python *cProfile*[6], the author analyzed the distribution of elapsed times within different code blocks. The performance issue was traced back to a function sorting long spike trains too slowly. To improve the performance of the relevant source code, the time critical section was rewritten using C++. An easy approach for optimizing small snippets of Python code is `scipy.weave`[7] [*SciPy*, 2008]. Table 5.2 shows typical runtimes for different functions using the unoptimized (upper half) or optimized (lower half) code section. The optimized function `setInput` achieves a speedup of 20, the total runtime was reduced by a factor of 3 for the given setup[8].

5.4 WinDriver

Running large network experiments on a neuromorphic hardware device generates large amounts of data. For an efficient data analysis, a modern computer system is desirable. However, as computer development advances, new technologies are being integrated into off-the-shelf computers. But new hardware requires new drivers and state-of-the-art operating systems. Some of the hardware to access the *FACETS Stage 1* system was developed in 2001. In particular, the host interface is implemented as a PCI[9] card developed within the author's group [*Becker*, 2001]. A proprietary driver (*WinDriver* by *Jungo Ltd* [2007]) provides a userspace interface which the low-level software framework utilizes. As the *Linux* kernel driver interface does not provide any stable API[10], drivers being out of the kernel tree tend to stop working after some releases. The same holds for the *WinDriver* module. The author adapted the utilized *WinDriver* kernel module several times to newer kernel versions. After installing a new lab computer the necessity to use Linux kernel version 2.6.24 arose. This kernel version changes the traditional *scatterlist* structures which are used in various DMA[11] operations. A new so-called *chained scatterlist API*[12] was introduced. The author modified the driver source code to use the new API.

[6] `http://docs.python.org/lib/module-profile.html`

[7] Weave is a subpackage of SciPy, which itself is a package for scientific computing that uses the Python programming language. Using *OpenMP*, Weave can be parallelized (GCC $\geq$ 4.2.4).

[8] Coincidence detector: simulation time was 10 s, 100 synapses with $1 \cdots 14$ Hz input rate each (in biological units). See *Kaplan* [2008].

[9] Peripheral Component Interconnect

[10] `http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt`

[11] Direct Memory Access

[12] `http://lwn.net/Articles/256368/`

calls	t_{total} [s]	filename:linenumber	function
1	88.52 ± 0.99	`pyhal.py:289`	`setInput`
1	12.50 ± 0.99	`pyhal_config.py:638`	`run`
3451471	6.03 ± 0.09	`__init__.py:634`	`lambda x,y: fcmp(x[0],y[0])`
12	3.59 ± 0.02		*method 'sort' of 'list' objects*
3451471	3.29 ± 0.09	`__init__.py:144`	`fcmp`
202	2.70 ± 0.30		*numpy.core.multiarray.array*
1	1.53 ± 0.02	`__init__.py:590`	`generateHardwareSpikeTrain`
2	0.78 ± 0.00	`pyhal_config.py:605`	`applyConfig`
1	0.62 ± 0.02	`__init__.py:329`	`run`
390607	0.38 ± 0.01	`pyhal.py:410`	`startOffset`
...	...	...	...
7460957	$\mathbf{121.80 \pm 1.68}$	**full run**	

(a) unoptimized

calls	t_{total} [s]	filename:linenumber	function
1	15.19 ± 1.95	`pyhal_config.py:638`	`run`
3451471	6.10 ± 0.04	`__init__.py:634`	`lambda x,y: fcmp(x[0],y[0])`
1	4.24 ± 1.11	`pyhal.py:289`	`setInput`
14	3.61 ± 0.05		*method 'sort' of 'list' objects*
3451471	3.15 ± 0.04	`__init__.py:144`	`fcmp`
202	2.49 ± 0.02		*numpy.core.multiarray.array*
1	1.55 ± 0.03	`__init__.py:590`	`generateHardwareSpikeTrain`
2	0.78 ± 0.01	`pyhal_config.py:605`	`applyConfig`
1	0.62 ± 0.01	`__init__.py:329`	`run`
17632	0.29 ± 0.01	`arrayprint.py:192`	`array2string`
...	...	...	...
7135199	$\mathbf{39.99 \pm 1.67}$	**full run**	

(b) `setInput()` optimized.

Table 5.2: Profiler output sorted by total time spent in function. 10 largest contributors shown.

6 Discussion & Outlook

The robust operation of a neuromorphic hardware device is a challenging task. Identifying error sources in such complex and novel hardware systems is difficult, however, at the same time, essential to provide feedback to hardware developers, allowing them to fix design flaws. Additionally, designing software environments for future users and developers to build upon is equally important and worthwhile.

In the course of this work multiple hardware malfunctions were found. After diagnosing these bugs, workarounds have been developed to provide a solution for the existing systems, since redesigns cannot be expected to be available within short terms and might still be faulty.

The readout deadlock of neurons was investigated. Groups of neurons available for readout were found: ca. 50 neurons can be read out per chip at the same time. A workaround resetting both neurons and the readout repetitively has been developed together with Dr. A. Grübl. Unfortunately, a bug in low-level code outside of the author's domain prevents the usage of the promising workaround.

The stability and correctness of parameters generated on-chip was inspected. Subsequently, an indirect measuring method was developed to verify write operations of the parameter values on-chip. Using this method, optimal write times were determined. An algorithm optimizing the writing order of parameters was designed and put into successful operation on *Spikey 2*.

Measurements to specify *STDP* functionality were carried out and have revealed various problems, especially the impossibility to find a consistent configuration which works for all synapses. Within the temporal scope of this diploma thesis a fully conclusive specification of the problems could not be achieved.

All presented measurements and resulting workarounds were essential steps towards a useful and well controllable experiment framework based on the FACETS hardware system.

Further contributions were made improving the software work flow within the group. We aim for integrating our *PyNN.hardware* module into the open-source *PyNN* project. Open-source software is in the spirit of science: progress is shared, reviewed, criticized and transparent.

For us, establishing connections to open-source projects is desirable as it reduces redundant work. Another favorable effect is the broadening of standardized tools and methods.

The other way around, various open-source tools were integrated in our software framework facilitating and improving the software flow. First, as developing software is an iterative process, a fast build process is desirable. This process has been improved by applying a *compiler cache*. Second, as users start employing a complex software suite – like the *PyNN.hardware* module – so far unknown bugs may be detected. Multiple developers work within the same software project and might commit conflicting changes. In these cases a centralized tool to report, discuss and handle fixes or other solutions is beneficial. This has been set up in terms of *Trac* – a source code management tool.

Established software was enhanced, ported to newer operating system environments or extended. The so-called *WinDriver* kernel module has been adapted to work with newer

Linux kernels (2.6.24). Various low-level hardware tests were documented and integrated into the *PyNN.hardware* framework. Some code profiling was done on the *PyHAL* layer yielding improvements in long-running experiments. To allow for a more robust calibration of the parameter voltages generated on-chip, a new routine has been developed.

A GUI for intuition-guided manipulating of experimental parameters and graphical representation of experimental data in real-time was developed.

These rather disjoint pieces of optimization represent the effort of aiming at a consistently well documented and well chosen set of harmonizing tools. Software should not be an additional source of complexity and errors as hardware itself is complex enough.

A possible future *Stage 1* version with several improvements implemented, could solve many problems described in this thesis. First of all, a fixed neuron readout bug would eventually permit larger experimental setups. However, as the existing connection to and from the *Spikey* chip is at the moment neither fast nor robust, two changes would be helpful, too: a decreased speedup factor (from 10^5 to 10^4 with regard to biological real-time) and a Gigabit Ethernet connection replacing the old, electrically error-prone SCSI-like connection. A decreased speedup would pose several advantages: first, the biological bandwidth would increase by the same factor by which the speedup would decrease. This would allow for a larger margin to higher biological firing rates. The present speedup and connectivity support an average rate of 10 Hz Poisson firing – a tenfold decrease in speedup would enable 100 Hz Poisson firing, resulting in a major difference in many experimental setups. Second, relative mismatch of circuit components would decrease, which could facilitate all calibrations. The most important change which will allow such a decrease in speedup would be an enlarged membrane capacitance. As the present conductances controlling the membrane voltage are typically too large, the time constant of the membrane is too small. A reduced speedup without an enlarged membrane capacitance would make this problem even worse.

To increase the *STDP* time constants modifications of transistor geometry are probably needed. However, these are likely to lead to further changes in the behavior in the chip which are difficult to predict at the moment. Possibly this could also solve some of the erratic behavior seen in previous sections.

The *FACETS Stage 2* hardware system will raise challenging tasks. The high number of neurons and synapses requires large amounts of configuration data – as the input and output data streams have to transport up to 1 TeraEvent/s during runtime. Complex biological networks have to be mapped onto the hardware system using the previously introduced *Graph Model*. However, optimizing the mapping process and data flow to and from the system will be inevitable. In order to avoid bottlenecks which massively affect overall system performance, experiments have to run asynchronously, i.e. pipelined. During experiments, new configuration, in- and output data have to be transmitted. Thus, all parts of the software framework taking part in the configuration process, spike train generation and output data analysis have to be parallelized. A future connection via HyperTransport allowing for real-time interaction between software and neuromorphic hardware will impose further constrains on the software environment.

The last year's progress in the development of neuromorphic hardware encourage continuing our efforts, which will most certainly yield major contributions to the spectrum of neural network simulators.

A Appendix

A.1 Source Code Locations

The following sections contain links to internal (FACETS/Electronic Vision(s) Group) source code repositories. For external access please contact the author: `mailto:mueller@kip.uni-heidelberg.de`.

A.1.1 General

- PyNN.hardware/PyHAL Source Code:
  - `http://kip1.kip.uni-heidelberg.de/repos/FACETSHDDD/software/trunk/src/hardware/stage1/`
- Low-Level C++ Code:
  - `http://kip1.kip.uni-heidelberg.de/repos/VISION/project/spikey/`

A.1.2 Further Contributions

- Automatic Low-Level Tests:
  - `http://kip1.kip.uni-heidelberg.de/repos/FACETSHDDD/software/trunk/src/hardware/stage1/pyhal/pyhal_hwtest.py`
  - `http://kip1.kip.uni-heidelberg.de/repos/FACETSHDDD/software/trunk/src/hardware/stage1/pyhal/wrappers/pyhwtest.h`
  - `http://kip1.kip.uni-heidelberg.de/repos/FACETSHDDD/software/trunk/src/hardware/stage1/pyhal/wrappers/pyhwtest.cpp`
- Neuron Resets:
  - `http://kip1.kip.uni-heidelberg.de/repos/VISION/project/spikey/src/tb/sc_sctrl.cpp`
- Parameter RAM Update Optimization:
  - `http://kip1.kip.uni-heidelberg.de/repos/VISION/project/spikey/src/tb/genTimeLut.py`
- Parameter Voltage Calibration:
  - `http://kip1.kip.uni-heidelberg.de/repos/VISION/project/spikey/src/tb/spikeyvoutcalib.h`
  - `http://kip1.kip.uni-heidelberg.de/repos/VISION/project/spikey/src/tb/spikeyvoutcalib.cpp`
- Build Process:

  - http://kip1.kip.uni-heidelberg.de/repos/FACETSHDDD/software/trunk/src/hardware/stage1/pyhal/Makefile

- Source Code Management (*Trac*):
  - https://cetares.kip.uni-heidelberg.de/cgi-bin/trac.cgi
- KoreScope:
  - http://kip1.kip.uni-heidelberg.de/repos/VISION/project/facets/scripts/python/tools/KoreScope/
- WinDriver:
  - https://cetares.kip.uni-heidelberg.de/cgi-bin/trac.cgi/wiki/WinDriver

A.1.3 Documentation

A link to the full documentation for the PyNN.hardware module and all submodules follows:

- http://kip1.kip.uni-heidelberg.de/repos/VISION/user/mueller/Documentation_PyNN.hardware_20080909/

A.2 Example: KoreScope

Listing A.1: PyNN script with KoreScope interface

```
import time, math

x,y = range(0, 1024, 1), range(0, 1024, 1)
frequency, amplitude, phase = 1.0, 1.0, 0.0

def work(): # pyNN.run()
    for i in xrange(1):
        for i in xrange(2):
            time.sleep(0.1)
            print '.',
            sys.stdout.flush()
        print ''
        for j in xrange(len(y)):
            y[j] = math.sin(1/frequency * j/(2*math.pi) \
                   + phase/(2*math.pi)) * amplitude

# update variables from gui (callback functions)
def updateAmplitude(value):
    global amplitude
    amplitude = value
def updateFrequency(value):
    global frequency
    frequency = value
def updatePhase(value):
    global phase
    phase = value

# register some sliders
CSwin.registerSlider(updateAmplitude, 0, 10)
CSwin.registerSlider(updateFrequency, 1, 100, 50)
CSwin.registerSlider(updatePhase)

def CSrun(): # obligatory function, called by gui
    work()
```

A.3 SciPy.Weave Example

Listing A.2: Code snippet showing SciPy.Weave usage for a time critical section.

```
    # try to use SciPy.Weave
    try:
        # using a C++ for-loop
        import scipy.weave as weave
        numPresyns = int(conf.numPresyns)
        numExternalInputs = int(conf.numExternalInputs)
        code = \
"""
for (int i = 0; i < oldsize; ++i) {
    // determining synapse driver for this input
    myC(1,i) = myC(1,i)/numPresyns * numPresyns +
                numPresync - 1 - (myC(1,i) % numPresyns);
}
"""
        weave.inline(code, ['oldsize', 'myC', 'numPresyns'],
                     type_converters = weave.converters.blitz)

    # using Python as SciPy.Weave did not work
    except:
        for i in xrange(oldsize):
            index = myC[1][i]
            if index > conf.numExternalInputs:
                raise 'Index of external input source too high!'
            # determining synapse driver for this input
            blockoffset = int(index/conf.numPresyns)*conf.numPresyns
            syndriver = blockoffset + conf.numPresyns - 1 -
                        (index%conf.numPresyns)
            myC[1][i] = syndriver
```

List of Abbreviations

ANNCORE	Analog Neural Network Core
CMOS	Complementary Metal-Oxide-Semiconductor
DAC	A Digital to Analog Converter converts a digital value to an analog signal.
DMA	Direct Memory Access allows certain computer subsystems to access system memory independently of the CPU.
EPSP	Excitatory Post-synaptic Potential
FACETS	Fast Analog Computing with Emergent Transient States
FIFO	First-In First-Out
FPGA	Field-programmable gate array
GbE	Gigabit Ethernet, IEEE 802.3-2005. Transmitting Ethernet frames at a rate of a gigabit per second.
GiB	1 GibiByte refers to 2^{30} bytes; close to 1 gigabyte ($= 10^9$ bytes).
GUI	Graphical User Interface
HICANN	High Input Count Analog Neural Network
LIF	Leaky Integrate-and-Fire
MD4	message digest algorithm (implements a cryptographic hash function)
NEST	NEural Simulation Tool
PSP	Post-synaptic Potential
PyHAL	Python Hardware Abstraction Layer
PyNN	Python package for simulator-independent specification of Neuronal Network models
RAM	Random Access Memory – a type of computer data storage
SCSI	Small Computer System Interface, standard for physical connection data transfer between computers and peripheral devices.
StdErr	Standard Error; default stream where error messages are written to
StdOut	Standard Output; default stream where output is printed to
STDP	Spike timing dependent plasticity
SVN	Subversion is a version control system and de-facto successor to Concurrent Versions System (CVS)
TCP/IP	Internet Protocol Suite including the Transmission Control Protocol (TCP) and the Internet Protocol (IP)
VLSI	Very-Large-Scale Integration
Vout	Parameter voltages generated on-chip are called *vout*s.

Bibliography

Becker, J., Ein FPGA-basiertes Testsystem für gemischt analog/digitale ASICs, Diploma thesis (german), University of Heidelberg, HD-KIP-01-11, 2001.

Bi, G., and M. Poo, Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type, *Neural Computation*, *9*, 503–514, 1997.

Bill, J., Diploma thesis, Diploma thesis, University of Heidelberg, presumably published in late 2008, 2008.

Brette, R., and W. Gerstner, Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity, *J. Neurophysiol.*, *94*, 3637 – 3642, 2005, article.

Brette, R., and D. Goodman, Brian, 2008, a simulator for spiking neural networks based on Python.

Brette, R., et al., Simulation of networks of spiking neurons: A review of tools and strategies, 2006.

Brüderle, D., Implementing spike-based computation on a hardware perceptron, Master's thesis, Heidelberg University, 2004.

Brüderle, D., A. Grübl, K. Meier, E. Mueller, and J. Schemmel, A software framework for tuning the dynamics of neuromorphic silicon towards biology, in *Proceedings of the 2007 International Work-Conference on Artificial Neural Networks (IWANN'07)*, vol. LNCS 4507, pp. 479–486, Springer Verlag, 2007.

Colclough, M., and G. Vermeulen, PyQwt – a set of Python bindings for the Qwt C++ class library, `http://pyqwt.sourceforge.net/`, 2007.

Dally, W. J., and J. W. Poulton, *Digital systems engineering*, Cambridge University Press, New York, NY, USA, 1998.

Dayan, P., and L. F. Abott, *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*, The MIT press, Cambride, Massachusetts, London, England, 2001.

Destexhe, A., Conductance-based integrate-and-fire models, *Neural Comput.*, *9*, 503–514, 1997.

Destexhe, A., M. Rudolph, and D. Pare, The high-conductance state of neocortical neurons in vivo, *Nature Reviews Neuroscience*, *4*, 739–751, 2003.

Ehrlich, M., K. Wendt, and R. Schüffny, Parallel mapping algorithms for a novel mapping & configuration software for the facets project, in *CEA'08: Proceedings of the 2nd WSEAS International Conference on Computer Engineering and Applications*, pp. 152–157, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2008.

Ensemble, N., PyNN – a python package for simulator-independent specification of neuronal network models, `http://www.neuralensemble.org/trac/PyNN`, 2008.

Fieres, J., J. Schemmel, and K. Meier, Realizing biological spiking network models in a configurable wafer-scale hardware system, in *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008.

Frey, U., et al., Cell recordings with a cmos high-density microelectrode array, in *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pp. 167–170, Lyon, 2007.

Grübl, A., VLSI implementation of a spiking neural network, Ph.D. thesis, Ruprecht-Karls-University, Heidelberg, 2007, document No. HD-KIP 07-10.

Hafliger, P., Adaptive WTA with an analog VLSI neuromorphic learning chip, *IEEE Transactions on Neural Networks*, *18*, 551–72, 2007.

Hines, M., J. W. Moore, and T. Carnevale, Neuron, 2008.

Hodgkin, A. L., and A. F. Huxley, A quantitative description of membrane current and its application to conduction and excitation in nerve., *J Physiol*, *117*, 500–544, 1952.

Huettel, S. A., A. W. Song, and G. McCarthy, *Functional Magnetic Resonance Imaging*, Sinauer Associates, 2004.

IEEE, Standard for information technology - portable operating system interface (POSIX). shell and utilities, *Tech. rep.*, IEEE, 2004.

Jungo Ltd, *WinDriver*, 1 Hamachshev Street, P.O.Box 8493, Netanya 42504, Israel, 2007.

Kandel, E. R., J. H. Schwartz, and T. M. Jessell, *Principles of Neural Science*, 4 ed., McGraw-Hill, New York, 2000.

Kaplan, B., Preliminary working title: Tuning the dynamics of a highly accelerated neuromorphic hardware towards biology and exploiting its speed for systematic self-organisation experiments, Diploma thesis, University of Heidelberg, presumably published in late 2008, 2008.

Kerr, J. N., D. Greenberg, and F. Helmchen, Imaging input and output of neocortical networks in vivo., *Proc Natl Acad Sci U S A*, *102*, 14,063–14,068, 2005.

Kumar, A., S. Schrader, A. Aertsen, and S. Rotter, The high-conductance state of cortical networks, *Neural Computation*, *20*, 1–43, 2008.

Maass, W., Networks of spiking neurons: the third generation of neural network models, *Neural Networks*, *10*, 1659–1671, 1997.

Markram, H., Y. Wang, and M. Tsodyks, Differential signaling via the same axon of neocortical pyramidal neurons., *Proceedings of the National Academy of Sciences of the United States of America*, *95*, 5323–5328, 1998.

McCulloch, W. S., and W. Pitts, A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics*, pp. 127–147, 1943.

Mead, C. A., *Analog VLSI and Neural Systems*, Addison Wesley, Reading, MA, 1989.

Mead, C. A., and M. A. Mahowald, A silicon model of early visual processing, *Neural Networks*, *1*, 91–97, 1988.

Mehring, C., J. Rickert, E. Vaadia, S. C. de Oliveira, A. Aertsen, and S. Rotter, Inference of hand movements from local field potentials in monkey motor cortex, *Nat. Neurosci.*, *6*, 1253–1254, 2003.

Merolla, P. A., and K. Boahen, Dynamic computation in a recurrent network of heterogeneous silicon neurons, in *Proceedings of the 2006 IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, 2006.

Morrison, Abigail, Diesmann, Markus, Gerstner, and Wulfram, Phenomenological models of synaptic plasticity based on spike timing, *Biological Cybernetics*, *98*, 459–478, 2008.

Ostendorf, B., Charakterisierung eines Neuronalen Netzwerk-Chips, Diploma thesis (german), University of Heidelberg, HD-KIP 07-12, 2007.

Pissanetzky, S., *Sparse Matrix Technology*, Academic Press, London, 1984.

Renaud, S., J. Tomas, Y. Bornat, A. Daouzli, and S. Saïghi, Neuromimetic ics with analog cores: an alternative for simulating spiking neural networks, in *Proceedings of the 2007 IEEE Symposium on Circuits and Systems (ISCAS2007)*, 2007.

Riverbank Computing Limited, PyQt – Python bindings for Trolltech's Qt application framework, `http://www.riverbankcomputing.co.uk/software/pyqt/intro`, 2007.

Rosenblatt, F., The perceptron: a probabilistic model for information storage and organization in the brain, *Psychological Review*, *65*, 386–408, 1958.

Schemmel, J., WP7 STDP implementation, 2006, university of Heidelberg.

Schemmel, J., personal communication, 2008.

Schemmel, J., K. Meier, and E. Mueller, A new VLSI model of neural microcircuits including spike time dependent plasticity, in *Proceedings of the 2004 International Joint Conference on Neural Networks (IJCNN'04)*, pp. 1711–1716, IEEE Press, 2004.

Schemmel, J., A. Grübl, K. Meier, and E. Mueller, Implementing synaptic plasticity in a VLSI spiking neural network model, in *Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN'06)*, IEEE Press, 2006.

Schemmel, J., D. Brüderle, K. Meier, and B. Ostendorf, Modeling synaptic plasticity within networks of highly accelerated I&F neurons, in *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS'07)*, IEEE Press, 2007.

Schemmel, J., J. Fieres, and K. Meier, Wafer-scale integration of analog neural networks, in *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*, 2008.

SciPy, Weave, `http://www.scipy.org/Weave`, 2008.

Serrano-Gotarredona, R., M. Oster, P. Lichtsteiner, A. Linares-Barranco, R. Paz-Vicente, F. Gomez-Rodriguez, H. K. Riis, T. Delbrück, and S.-C. Liu, AER building blocks for multi-layer multi-chip neuromorphic vision systems, in *Advances in Neural Information Processing Systems 18*, edited by Y. Weiss, B. Schölkopf, and J. Platt, pp. 1217–1224, MIT Press, Cambridge, MA, 2006.

Song, S., K. Miller, and L. Abbott, Competitive hebbian learning through spiketiming-dependent synaptic plasticity, *Nat. Neurosci.*, *3*, 919–926, 2000.

Stroustrup, B., *The C++ Programming Language*, Addison Wesley, Reading, MA, 1997.

The Neural Simulation Technology (NEST) Initiative, Homepage, `http://www.nest-initiative.org`, 2007.

Trolltech, Qt cross-platform application framework 4.2.0, `http://trolltech.com/developer/resources/notes/changes/changes-4.2.0/`, 2006.

Tsodyks, M., and H. Markram, The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability, *Proceedings of the national academy of science USA*, *94*, 719–723, 1997.

Vogelstein, R. J., U. Mallik, J. T. Vogelstein, and G. Cauwenberghs, Dynamically reconfigurable silicon array of spiking neuron with conductance-based synapses, *IEEE Transactions on Neural Networks*, *18*, 253–265, 2007.

Wendt, K., M. Ehrlich, and R. Schüffny, A graph theoretical approach for a multistep mapping software for the facets project, in *CEA'08: Proceedings of the 2nd WSEAS International Conference on Computer Engineering and Applications*, pp. 189–194, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2008.

Index

Boost.Python, 16, 21
Brian, *see* PyNN
Build Process, 37

Ccache, 37
Code Profiling, 39

Event Loopback Test, *see* Tests

FACETS Hardware Stage 1, 42
FACETS Hardware Stage 2, 42
FACETS Stage 1, 5
FACETS Stage 2, 10

Graph Model, 16

Integrate-and-Fire, *see* Neuron

Korescope, 38

Link Test, *see* Tests
Low-level Software, 15
LTP, *see* Plasticity

Makefile, 37

NEST, *see* PyNN
NEURON, *see* PyNN
Neuron
    Model, 7
    Readout Deadlocks, 22
    Resets, 28
Noise, 22

Parameter RAM
    Test, *see* Tests
    Update, 32
Plasticity
    LTP, 8
    STDP, 8
    STP, 7
    Synaptic Plasticity, 7
PyHAL, 13
PyNN, 11
    Brian, 13
    hardware, 13
    MOOSE, 13
    NEST, 12
    NEURON, 12
    PCSIM, 13
PyNN.hardware, *see* PyNN

SciPy.Weave, *see* Code Profiling
SCM, 37
Software Framework, 11
Spikey, 5, 6
STDP, *see* Plasticity
    seePlasticity, 42
STP, *see* Plasticity
Synaptic Plasticity, *see* Plasticity

Tests
    Event Loopback, 21
    Link, 19
    Low-level, 19
    Parameter RAM, 21
Trac, 37

Vout Calibration, 35

WinDriver, 39, 41

gggqG, 53

Acknowledgments (Danksagungen)

Ich möchte mich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben. Dies sind insbesondere:

Herrn Prof. Dr. Karlheinz Meier für die freundliche Aufnahme in die Arbeitsgruppe und die stets hilfreiche Unterstützung.

Herrn Prof. Dr. Thomas Ludwig für die Übernahme der Zweitkorrektur.

Daniel Brüderle für die intensive und hilfreiche Betreuung.

Bernhard Kaplan, Daniel Brüderle, Johannes Bill, Mihai Petrovici und Olivier Jolly – auch wenn er mittlerweile Vollblut Hardy ist – für die persönliche Büroatmosphäre.

Dres. Grübl & Schemmel für die Antworten auf meine vielen Fragen zur Hardware.

Vim & xJump

Allen Visionären für Hilfsbereitschaft und stets angenehme und freundschaftliche Arbeitsatmosphäre.

Meiner Familie für allumfassende Unterstützung.

Dem M. C. für zackige Mittagessen.

Allen Korrekturlesern – insbesondere Börni, Daniel, Johannes, Kathi, Maike und Mihai.

Brutal B, Dirty D, Jigsaw J, Merdi, Mighty M und Olivier für Spaß (& Frust!) am Kickertisch.

Katharina.

`:xa!`

Statement of Originality (Erklärung):

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, September 10, 2008

.....
(signature)